Optimised Architectures and Implementations for Efficient Neuromorphic Hardware Design

Graham-Harper-Cater, Jonathan

Award date:
2019

Awarding institution:
University of Bath

Optimised Architectures and Implementations for Efficient Neuromorphic Hardware Design

submitted by

Jonathan Edward Graham-Harper-Cater

for the degree of Doctor of Philosophy

University of Bath

Department of Electronic and Electrical Engineering

July, 2019

Trust in the Lord with all your heart
and lean not on your own understanding;
in all your ways acknowledge him,
and he will make your paths straight.
Proverbs 3v5

Acknowledgements

This work would never have been possible without the continued help and support of a great many people over the last three years. While the list of individuals to whom I am truly grateful could easily fill multiple pages within this text, there are a handful of kind people, highlighted here, that are especially worthy of mention.

Firstly, I would like to thank Dr Benjamin Metcalfe, my primary supervisor, alongside Prof. Peter Wilson and Dr Chris Clarke. Their advice, guidance and supervision contributed significantly to the completion of this work, and they have continuously offered timely support and constructive criticism throughout this period. I am truly grateful for their advice and friendship.

I would also like to thank Eur. Ing. Dr Brian Nicholson QC, who generously sponsored my doctoral study, showing unwavering support and interest in my work while asking for little in return. I am eternally grateful for this act of kindness, and take great encouragement that somebody would support the academic efforts of another individual so freely.

I am thankful for the whole family at both St Matts, Widcombe and Trinity Baptist Church, Gloucester, whose ceaseless prayer and support has strengthened me throughout the last three years. It was a pleasure to study God's Word with the members of the Holloway small-group, led by Steve and Beth Forrester. It has been an honour to walk alongside each of these kind individuals and it is my prayer that God may bless them in the years to come.

I must also thank my colleges and friends Dr Alex Beasley and Kye Pearce-Rees, whose light-hearted conversation and jokes helped brighten the office on the more challenging days and I am also grateful to Jonathan Harris for the many conversations over coffee.

Finally, I would like to thank my parents, Niall and Heather, as well as my brother Timothy. They have always encouraged and supported me without question, despite the many weeks spent hidden away writing this thesis.

Abstract

Computational processing is a critical element of modern-day life, enabling and enhancing research, industry, medical and military efforts. The continued improvements seen in processing speed and power are largely driven by the iterative scaling of transistors and other integrated circuit components. As predicted by Moore's law, this trend has resulted in processing solutions approximately doubling in computational power every two years. This trend cannot continue indefinitely, and it is widely agreed that new designs are rapidly approaching the physical limits of transistor scaling. It is therefore necessary that novel processing technologies are developed to allow the continued advancement of processing potential. One promising research effort is that of neuromorphic computing, a field that takes inspiration for the design of new and efficient processing technologies from biological nervous systems. Neuromorphic systems excel at cognitive computational tasks that standard processing solutions typically find challenging. These systems therefore represent a crucial tool in the future of computational technologies. Despite significant advancements, neuromorphic solutions fall short of the efficiencies seen in nature and further research into such designs is therefore critical in ensuring their continued success and wider application.

This thesis is concerned with the impact of low-level element design on a neuromorphic system's efficiency and computational power. Significant improvements in both speed and efficiency are demonstrated, achieved by careful redefinition of the fundamental building blocks and structures common to neuromorphic solutions. The proposed systems developed in this work achieve this improved performance without reduction in computational function by redesigning the underpinning operations and implementations from the ground up with engineering constraints and principles in mind. The gains demonstrated with this approach are shown to benefit both biophysically accurate and computationally efficient neural models, offering further acceleration on existing neuromorphic architectures. Alongside the fundamental building blocks, the connection infrastructure common to modern neuromorphic solutions is also considered, showing that there is a considerable difference between hardware implementations and biological systems. This difference in structure and applied connectivity appears largely due to a fundamental difference in the dimensionality available in each case, with hardware systems commonly constrained to a low number of two-dimensional layers, while biological systems form dense three-dimensional structures. The results of this work show the critical role that function and system definition plays in the efficiency and computational power of neuromorphic systems. Through the application of these findings, future neuromorphic systems can achieve greater performance-per-watt with reduced computational delays.

Contents

List of Figures

List of Tables

List of Abbreviations

ALM	Adaptive Logic Modules
ALU	Arithmetic Logic Units
ALUT	Adaptive Look-Up Table
ANN	Artificial Neural Network
AP	Action Potential
ASIC	Application Specific Integrated Circuit
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
CNN	Convolutional Neural Network
CNS	Central Nervous System
CPG	Central Pattern Generator
CPU	Central Processing Unit
DARPA	Defence Advanced Research Projects Agency
DNN	Deep Neural Network
DPI	Differential Pair Integrator
DRC	Design Rule Checking
DSP	Digital Signal Processing
FF	Flip Flop

FFNN Feed-Forward Neural Network

FIFO First-In, First-Out

FPGA Field Programmable Gate Array

GALS Globally Asynchronous, Locally Synchronous

GP Graded Potential

GPU Graphical Processing Unit

GUI Graphical User Interface

HDL Hardware Description Language

HNS Human Nervous System

HPC High Performance Computing

IC Integrated Circuit

IF Integrate and Fire

IIR Infinite Impulse Response

IoT Internet of Things

LSTM Long Short-Term Memory

LUT Look-Up Table

LVS Layout Versus Schematic

MLP Multi-Layer Perceptron

MRI Magnetic Resonance Imaging

NEST NEural Simulation Technology

NoC Network-on-Chip

ODE Ordinary Differential Equation

OPS Operations Per Second

PCIe	Peripheral Component Interconnect Express
PNS	Peripheral Nervous System
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SMU	Source Measurement Unit
TPU	Tensor Processing Unit
ULP	Unit in the Last Place
VLSI	Very-Large Scale Integration

Chapter 1

Introduction

The computer is one of the most disruptive innovations of the last two centuries and its adoption in commerce, industry and leisure has completely transformed the lives and livelihoods of people around the world. As computing systems have grown in both speed and power they have driven advancement in almost every aspect of life. Modern mobile phones can easily outperform the cutting-edge computers of the 1970's, and tasks that took months or even years of compute time, now take mere minutes on today's supercomputers. At the heart of these systems is the processor, responsible for the calculation and manipulation of data. These processors are now found in a wide variety of applications, from large scale data processing to low power mobile computing. Considering the common use of mobile phones, tablets, computers, digital car technologies and other appliances, it is reasonable to predict that processors now largely outnumber people in developed countries. With the adoption of 'Smart' technology and the 'Internet of Things', the integration of processors into every day tasks and operations will likely continue its accelerated trajectory for many years to come.

Since the first electronic processor there have been continued improvements in processor speeds and scales. In 1965, Gordon Moore observed that the number of components used in Integrated Circuits (ICs) approximately doubled each year [1], predicting that this rate of growth would continue. In 1975, Moore revised this prediction in recognition that the rate of expansion had slowed to doubling every two years [2]. This prediction, known as Moore's law, has proven relatively accurate and has therefore guided the semiconductor industry when setting long term plans or targets. The increase seen in component count is largely driven by the continued improvements in fabrication technologies, enabling smaller features such as transistors to be implemented on the ICs.

The benefit of this component scaling becomes clearer when taken in conjunction with Dennard scaling, which approximately states that the power density of transistors remains constant as their size is reduced [3]. As a result of these two laws, processors

have become approximately twice as computationally powerful for the same input power every two years. Alongside better power performance, smaller transistors also switch faster allowing each generation of processors to use faster clocks than the last.

The continued and reliable scaling of transistors has played a critical role in the advancement of processor technologies. Physical limits are rapidly approaching, however, and experts agree that Moore's law will soon reach its inevitable end. Transistor scales cannot be reduced beyond a few atoms without loss of function and Dennard scaling was observed to break down after 2006. As transistors have grown smaller the static power loss of ICs as a proportion of the overall power loss has increased. This is caused by increased leakage currents, quantum effects and changes to the chemical composition of modern ICs, each a direct result of implementing smaller transistor scales. These static power losses now play a key role in determining the performance of an IC meaning that the predictions made by Dennard's scaling no longer hold true [4]. Novel technologies are therefore required to continue the trend of increasing computational power as these laws breakdown.

There are many approaches that could yield improved processing capabilities over the conventional von Neumann architectures commonly found in a computers Central Processing Unit (CPU). Task specific processors, such as Graphical Processing Units (GPUs), offer an efficient and rapid computation solution for a reduced and well constrained problem set. GPUs, for example, are composed of hundreds of simplistic computational cores and can handle thousands of threads simultaneously. They are well suited for performing matrix oriented floating point operations and commonly find application in computer graphics, finance and machine learning. Performing general computing tasks that sit outside the scope of such processors becomes expensive in both time and energy. For this reason it is common for CPUs and GPUs to be used in conjunction with one another, with specific tasks handed off to the GPU to leverage its accelerated computation. It is possible to produce more efficient and rapid processing solutions as the problem set is further constrained. With each development, however, these task specific processors become more limited in their application making it necessary to complement them with other processing solutions. Additionally, the incremental improvements seen in these task specific processors are, in part, driven by the same component scaling as conventional processors and the end of Moore's law will therefore also impact the development of new task specific processors.

Cloud computing helps shift the computational load from lower-power or mobile devices to large-scale computing setups such as supercomputers or data centres. This enables mobile solutions to perform expensive and computationally challenging tasks but requires the device to be connected to the system via some network. Such off-site processing leads to security concerns when performing certain tasks making cloud computing unacceptable for military, government and certain high-value industrial applications. Even when acceptable, these large-scale data centres will have to grow significantly in scale to supply the increasing computational demand as the benefits from underlying processor scaling comes to a halt. At this point power consumption

rapidly becomes the dominant factor in deciding the upper limit on computational potential.

Research into novel materials, such as graphene, and new processing paradigms, such as quantum computing may help address the issues caused by the fundamental limits of silicon transistor scaling. Graphene has shown promising electrical characteristics, however application of these findings in the design of new and reliable components has proven a considerable challenge. Quantum computers could excel at a specific class of problems where a superposition of many possible solutions collapses into one discrete answer. The progress in this exciting field of computation is largely limited by the challenges associated with scaling quantum systems to support larger data-widths. As these quantum system are developed, researchers must also find novel ways to adapt and redefine their problems making them suitable for such systems.

In pursuit of greater power efficiency, 'neuromorphic computing' takes inspiration from nature, designing new processing paradigms and accelerating machine learning and Artificial Neural Network (ANN) applications. Neural networks have become ubiquitous as tools for data analysis and information processing, and their acceleration plays a key role in the future processing of large datasets. These networks have found widespread application in many research fields, alongside major integration into new commercial products where they typically excel at classification, speech generation and image recognition tasks [5]. State of the art results have been demonstrated that rival custom made and hand tuned algorithms and as these systems grow in scale they offer new solutions to previously challenging problems. With machine learning and neural networks becoming a core fundamental in computational applications, it is increasingly important to ensure that efficient and effective processors are developed that excel at these tasks.

To address this critically important global computing problem this thesis introduces new biologically inspired neuromorphic architectures that enable efficient computation, and neural models that support direct and effective hardware implementation.

1.1 Neuromorphic Computing

The term *neuromorphic* was originally introduced by Carver Mead in 1990 to describe circuits inspired by biological neural systems [6]. Mead drew parallels between the function and efficiency of neurons and transistors, arguing that the greater power efficiency seen in biological systems must therefore arise from differences in the application and utilisation of these fundamental building blocks. Mead went on to demonstrate a circuit that used the transistors fundamental non-linearities to perform computation in a manner similar to that of a biological neural network.

As the field has grown, the term neuromorphic computing has come to include any

system designed to accelerate or efficiently implement neuron models and neural networks in hardware. There has been considerable advancement in the application of GPUs for machine learning applications, however significant power savings and computational speed improvements may still be realised through the development and implementation of effective and specialised neuromorphic systems.

1.1.1 Motivation

The motivation for biologically inspired processors and hardware systems is largely based on observable differences between information processing seen in nature and that of data processing in electronic systems. While there are many benefits to be gained by learning from natural biological system, three core motivators may be identified as follows:

Power Consumption. Biological systems achieve considerable power efficiency, with the human brain estimated to use just 20 watts of power whilst modern processors use around 100W, and GPUs can use anywhere between 100W to 300W. With both mobile computing and large-scale computing becoming more pervasive, power savings in computation are financially and environmentally driven. Replicating the function of biological systems may allow new computational solutions to realise the high efficiencies seen in nature.

Cognitive Solutions. Since the early part of the 21st century, machine learning and artificial intelligence have seen considerable uptake in both commercial and industrial applications. These implementations depend heavily upon cognitive computing principles, where learned patterns and relationships guide the result generation. This cognitive processing differs significantly from conventional processing techniques, with cognitive systems capable of producing reasoned outputs rather than performing well-defined mathematical instructions. While conventional processors excel at tasks where sequential instructions and mathematical operations are required, cognitive systems excel in pattern recognition, classification and inference tasks. The results are often generated with a measure of uncertainty making the systems better at rapidly estimating a result, but less suitable for precise mathematical calculations where the input/output relationship or function has been fully defined.

Cognitive computing closely matches the way that humans process and interpret information and this similarity promises new and exciting opportunities in computing technologies. Future cognitive systems stand to revolutionise the way that individuals interact with computational devices, potentially resulting in systems capable of interpreting human intention and independently producing persuasive and compellingly reasoned results. These systems complement traditional processors, increasing the range of tasks that may be performed using artificial systems.

Learning and Adaption. Biological systems also demonstrate a unique ability to learn and adapt. As a result these systems are robust, demonstrating a high tolerance to noise, changing task parameters or localised damage. Reorganisation has been observed in animals, where the nervous system attempts to recover and preserve function after a stroke [7]. Brain plasticity has been associated with both recovery and learning, allowing new pathways to form in the brain to establish new or previously lost function [8]. In contrast, ICs usually represent constrained and heavily defined solutions that must rely on redundancy to achieve defect tolerance. Unlike biological neurons, these redundant elements are often designed for a specific task or role making them useless in the event of failure elsewhere on the IC.

Each of these features and properties collectively motivate the development of new neuromorphic systems. As machine learning and neural networks expand in both scale and application, neuromorphic systems will become critical elements in their delivery and success. With the correct setup and design, a neuromorphic system may one day achieve comparable performances to that of biological systems helping 'bridge the gap' left in the wake of Moore's law.

1.1.2 Challenges

There are many challenges and open questions that must be addressed to develop the next generation of neuromorphic systems. The major challenges may be classified under four points as follows:

Model Complexity. A critical juncture is the trade-off between the biophysical accuracy and computational efficiency of the neuron models. The biological neuron is a complex system that may by described using multiple non-linear Ordinary Differential Equations (ODEs). To accurately represent this system, models must contain complex non-linear behaviours with a large range of time constants. This makes such biophysically accurate models computationally expensive in both IC resource and time. Biophysical models are therefore impractical for large scale network implementations. On the other hand, computationally efficient models must balance model simplicity against their cognitive capability. If the model is too simple, the network may fail to fit the problem space in a useful manner. The design and selection of a suitable model complexity is therefore an important area in neuromorphic computation.

Physical Implementation. Models that perform well in a simulation may map poorly to hardware implementations. Equally, hardware implementations can offer accelerated and efficient operation while performing functions that are traditionally challenging on a standard processor. Software relies heavily on memory and abstraction allowing densely connected networks to be modelled and manipulated with relative ease. Hardware can accelerate the operation of

these networks and models but often loses the abstraction in doing so. Communications infrastructure and model synchronisation rapidly becomes a critical design consideration. Accelerated functions and operations must be selected carefully to reduce redundant elements within the hardware solution. These design decisions start to constrain the final models that are optimally supported by the hardware. The large difference in operation means that moving a neural system from simulation to physical implementation requires considerable time and engineering investment. It is therefore not possible to fully leverage the gains of a hardware system without first fully understanding the underlying infrastructure and support offered by the hardware. If a hardware implementation is intended, it is important to consider the final hardware implementation when defining the original neural models. This can ensure that any potential gain in power efficiency or speed is not compromised during the transfer from software to hardware.

Biophysical Interfacing. As neuromorphic systems have developed, new models of neurons have been made that closely represent the function and operation of their biological counterparts. Building systems that interface between hardware and biological neurons is now a feasible but challenging aim. Such systems stand to improve existing medical technologies, such as pace makers. The development of these systems requires considerable investigation to ensure that the final product is safe and reliable.

Increasing Scale. Neural networks have seen a resurgence in the early part of the 21st century, finding new applications in neurological, engineering and computing research and products. The networks used in these applications have seen continuous growth in scale or complexity. These networks are now reaching scales of several million neurons and billions of synapses and neuromorphic hardware must provide the means to efficiently and rapidly implement these networks for such trends to continue. The communications used to connect neurons together and the computation required by each neuron represent bottlenecks in this scaling trend and future systems must address these areas to support networks of greater scale.

Addressing one or many of these challenges is a key focus in modern neuromorphic research. As the system grow in scale the complexity and physical implementations become increasingly important as small changes to individual neurons can have compound effects on the whole system.

1.2 Contributions of this Thesis

The following are the major contributions of this thesis towards neuromorphic systems design:

1. Chapter 5 presents experimental results from tests of a biophysically accurate sub-threshold neural model implemented on $0.35\mu m$ fabrication technology, showing that, while a functional analogue neuron model is possible in this technology, it is highly susceptible to process variation. This custom IC was validated against SPICE Monte-Carlo simulations, demonstrating the relative value of sub-threshold simulations using SPICE. Results from tests in high magnetic fields of 3 Tesla are also provided, representing the first of many tests to validate the model for insertion into a next generation pace maker.

2. In Chapter 6 optimised approximation models for digital Hodgkin-Huxley based neurons were developed and tested. Good model representation is demonstrated using a two-stage hybrid model for an exponential operation that provides high levels of detail for small signal inputs while maintaining a constant maximum error for the full input range. This model is shown capable of producing biophysically plausible phasic spiking and tonic spiking Action Potentials (APs).

3. A new base-two sigmoidal activation function is introduced in Chapter 7 and shown to produce equivalent classification results to that of the common logistic sigmoid function. A conversion for networks to move between the original logistic sigmoid and base-two activation functions is provided allowing pre-trained systems to move between implementations without the requirement of further training. A hardware accelerator for this base-two activation function is described and shown to outperform modern processors in both speed and energy consumption.

4. A novel locally-connected architecture is introduced in Chapter 8 that 'closes the gap' in differences between the connectivity seen in nature and that of artificial systems. This architecture is demonstrated using an event-based *C. Elegans* locomotive model, yielding correct locomotive behaviour while offering a clock-speed improvement of between 3.5× and 17.5× that of a comparable but globally connected system. Building on this work, fundamental limitations in our current technology are demonstrated as a core constraining factor when attempting to replicate the connectivity (and therefore function) of biological neural systems in hardware. In particular, the dimensionally seen in traditional IC technologies falls short of that seen in nature. This is shown to require greater levels of communication infrastructure resulting in reduced power efficiency and considerable network scaling challenges.

A list of publications inspired by the work of this thesis is given below:

- J. E. Graham-Harper-Cater, B. W. Metcalfe and P. R. Wilson, "An analytical

comparison of locally-connected reconfigurable neural network architectures using a C.Elegans Locomotive Model." *Computers,* vol. 7, no. 3, 2018.

- J. E. Graham-Harper-Cater, C. T. Clarke, B. W. Metcalfe and P. R. Wilson, "A Reconfigurable Architecture for Implementing Locally Connected Neural Arrays" *Proceedings of the 2018 SAI Computing Conference,* 2018.

- P. R. Wilson, B. W. Metcalfe, J. E. Graham-Harper-Cater and J.A. Bailey, "A Reconfigurable Architecture for Real-Time Digital Simulation of Neurons" *2017 Intelligent Systems Conference (IntelliSys),* 2017.

Chapter 2

Neurons In Nature

The fields of neural networks and neuromorphic engineering draw considerable inspiration from the natural world. Many neuromorphic systems are used to simulate and explore the function of biological neural networks in order to understand more about their functionality. It is essential therefore to have a good understanding of biological neural networks and their underlying principles when considering the design of neuromorphic systems and neural architectures. This understanding allows designers to develop new systems that emulate the amazing efficiency and cognitive power demonstrated by biological neural networks. Additionally, systems that accurately emulate biology may be used to expand the understanding of biological network dynamics, furthering fundamental neuroscience research.

This chapter provides an overview of the structure and operation of biological neural networks, taking the Human Nervous System (HNS) as an example. The underlying properties and functions of the individual neurons is then explored in greater detail in Section 2.2. The considerations when modelling neurons are identified in Section 2.3 before two key neuron models are described in Sections 2.4 and 2.5. Finally Section 2.6 reviews the simulation tools available for neuroscience research when utilising biophysically accurate neuron models.

The intention of this chapter is to introduce the underlying concepts pertinent to the remainder of this thesis. It should not, therefore, be considered an exhaustive description of the current scientific understanding of the HNS. For wider detail on the structure and function of the nervous system as a whole, the reader is advised to see Gray's Anatomy, chapter 8 [9].

2.1 The Human Nervous System

The HNS is a highly complex system, containing over 86 billion interconnected neurons [10]. These neurons play a critical role in the support of life and are directly responsible for the complex decision making processes and other higher functions that make the HNS an enviable processing system.

While the nervous system is essentially continuous, it may be conceptually divided into two key parts according to function as shown in Figure 2.1. The Central Nervous System (CNS) is largely responsible for the collection and cognitive processing of information, while the Peripheral Nervous System (PNS) acts as the interface between the CNS and principle receptor and effector organs. This section is therefore divided into two distinct parts, considering the CNS and PNS as two separate elements, each with their own unique structure and function.

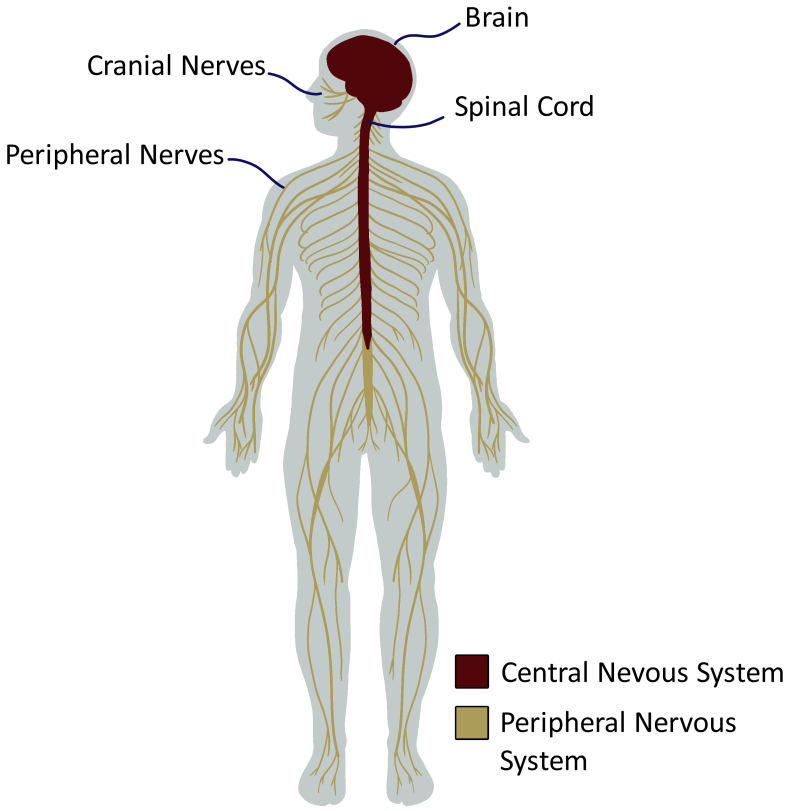


Figure 2.1: Example structure of the HNS, showing the high-level division of the CNS and PNS within the body.

It is important to note that there are many different ways to classify regions of the HNS. One alternative method uses a functional division, splitting the HNS into the *Somatic Nervous System*, that is associated with the control of body movement through skeletal muscles and involuntary reflex arcs; and the *Autonomic Nervous System*, that controls visceral functions that occur below the level of consciousness. In each case,

any division or classification used is defined to aid the communication of ideas and concepts within the research and medical community and does not actually represent physically distinct subsystems within the HNS.

2.1.1 The Central Nervous System

The CNS is primarily formed of the *encephalon* or brain and *medulla spinalis* or spinal cord and forms the primary component in processing and memory. The encephalon is further split into six lobes, each of which is responsible for a specific type or class of task. These lobes are: the frontal lobe, strongly tied to consciousness and largely responsible for thought, memory and behaviour; the parietal lobe, responsible for movement, language and touch; the temporal lobe, responsible for hearing, learning and emotions; the occipital lobe, responsible for visual processing; the cerebellum, responsible for balance and coordination; and the brain stem, responsible for breathing, heart rate and temperature management. The brain stem also plays the vital role of relaying information between the body and these other higher regions of the brain.

The cerebral cortex comprises of the outermost layers of the brain. Containing the frontal, parietal, temporal and occipital lobes, it is largely responsible for the processing of sensory information. The human cerebral cortex is formed as a highly folded sheet of neurons that has a thickness of about $1 - 4.5$mm yet yields a high surface area [11]. About half of its thickness consists of white matter, providing the dense connections between neurons contained within its structure [6]. It is the flat folded structure of the cerebral cortex that leads to the promising conclusion that somewhat 2-dimensional systems, such as that of modern Integrated Circuit (IC) fabrication processes, may produce systems capable of replicating functional elements of the CNS. Such replication of CNS structure and function on IC technology would have significant impact on the types of tasks that processors can perform, closing the gap between computational processing and human intuition.

2.1.2 The Peripheral Nervous System

The PNS is also formed of two key divisions, defined according to the direction of information propagation. The sensory, or afferent division, carries signals that 'enter' the brain and is responsible for receiving the stimuli generated by sensory nerve receptors. These receptors operate as specialised sensors or inputs, with each type responding to a specific stimuli, as shown in Table 2.1.

The motor, or efferent division, carries signals that 'exit' the brain, sending instructions to muscles and glands within the body. This division contains both the somatic, or voluntary nervous system, that rules skeletal muscle movement; and the autonomic, or involuntary nervous system, that controls internal organ function. While the CNS

Table 2.1: Common sensory nerve receptor types and their associated stimuli.

Nerve Receptor Type	Associated Stimulus
Thermoreceptors	Temperature
Photoreceptors	Visible light
Chemoreceptors	Chemical
Mechanoreceptors	Mechanical stress or strain
Nociceptors	Tissue damage

performs the bulk information processing, there are regions of the PNS, such as the sacral plexus, where processing of sensory inputs occurs somewhat independently to the CNS. These regions perform localised processing of information and control encapsulated bodily functions, such as digestion or bladder control. In other areas, some pre-processing of the sensory inputs occurs within the PNS prior to transmission through the remainder of the nervous system.

2.1.3 Summary

It is helpful to think of the HNS as two distinct sub-systems due to both its size and complexity. In this way the CNS provides bulk information processing and higher level functions associated with consciousness, while the PNS provides the interconnectivity within the body, and includes the sensory neurons used to detect a range of stimuli.

There are many parallels between that of the HNS and other engineered solutions. The localised specialisation seen within the CNS is not unlike standard computing systems, where a Central Processing Unit (CPU), volatile Random Access Memory (RAM) and a Graphical Processing Unit (GPU) all operate alongside one another to provide an effective and efficient system capable of general purpose computation. In engineering terms the PNS may also be thought of as the input and output interface for the nervous system. The brain stem and spinal cord is responsible for carrying signals to other parts of the body, and similarities may be drawn between this function and that of buses or other communications infrastructures seen in artificial systems.

While there is considerable published work exploring the HNS, it is widely recognised that further research is required to produce a comprehensive understanding of its finer structure and function. Such results may be found by starting with a high-level view of the HNS, iteratively refining the models and theories as the regions under consideration are further divided into sub-regions. Equally, one may begin with the smallest element within the system and attempt to reproduce its function and properties. These small elements may then be used as building blocks to form simulations of larger regions. These two approaches are very different in their methodology, however each stands to provide valuable insight into the others progress and findings of both must therefore be treated as important context for any neuroscience research efforts.

2.2 The Neuron

As identified in Section 2.1.3, the pursuit of improved computational power and performance has led to considerable interest in understanding the building blocks of the nervous system. Located throughout the entire HNS, the neuron is a fundamental component of the nervous system, playing a critical role in both the processing and communication of information. Neuronal morphology varies significantly throughout the different regions of the HNS, however all posses a set of common features and functions [12]. Figure 2.2 shows a myelinated neuron, highlighting the key features common to many neurons with the nervous system. The axons of myelinated neurons are additionally surrounded by *Schwann cells* that generate an insulating myelin sheath. All neurons contain a cell nucleus that is surrounded by cytoplasmic mass. This nucleus is located within the *soma*, a relatively large section of the the nerve from which fine filaments termed *neurites* extend. These neurites may be classified into one of two groups, the *dendrites* and the *axon*.

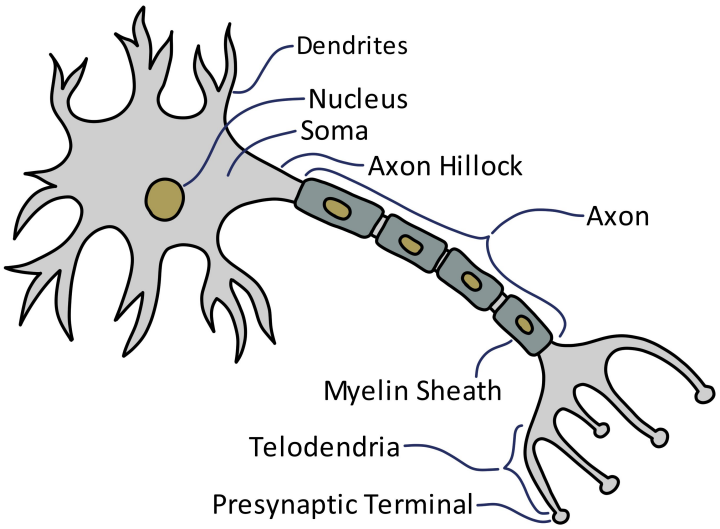


Figure 2.2: Structure of a generalised myelinated neuron.

The dendrites act as input channels for the neuron, propagating the incoming signals towards the soma. Equally, the axon begins at the axon hillock and operates as the output channel for the neuron, carrying an electrochemical pulse to the telodendria. Ranging from about $0.1 - 20\mu m$ in diameter, neurons typically have a singular axon. Information is transmitted along the axon as small all-or-nothing electrical impulses that propagate from the soma to the telodendria, called Action Potentials (APs).

While not common to all neurons, the axons of myelinated neurons are surrounded by a *myelin sheath*. The myelin sheath acts as an insulating layer, accelerating AP transmission while also helping to protect the axon filaments. Due to *salutatory conduction*, the propagating signals leap from one gap between a pair of myelin sheaths, termed the *nodes of Ranvier*, to the next. This greatly reduces the transmission time of

APs down the full length of long axons, with large peripheral nerves in humans capable of conducting APs at speeds up to 120 m/s.

Finally, the *telodendria*, located at the end of the axon, form synapses though electrochemical contact with the dendrites of other neurons. These synapses are not shown in Figure 2.2 but are located on the end of the telodendria. The synapses conduct the APs as electrochemical signals, producing input stimuli for the connected neurons. This input is then used within the next neuron to determine whether or not to produce the all-or-nothing AP. The relatively simple decision making process behind the generation of individual APs allows the nervous system to perform internal processing and data manipulation and in some ways may be compared to the application of binary logic gates within standard CPUs.

2.2.1 The Membrane Potential

These all-or-nothing APs are critical in the nervous systems ability to process information in an efficient and effective manner. To understand how neurons generate such APs, it is first important to consider the neurons membrane potential. All cells have a membrane potential as a result of differences in the cells internal and external charge, or ionic concentrations. The membrane provides a capacitive insulation layer between internal and external charges; neurons can therefore manipulate the membrane potential by controlling the flow of ions across the cell membrane itself. This *concentration gradient* forms a potential difference in the cell body relative to the surrounding medium.

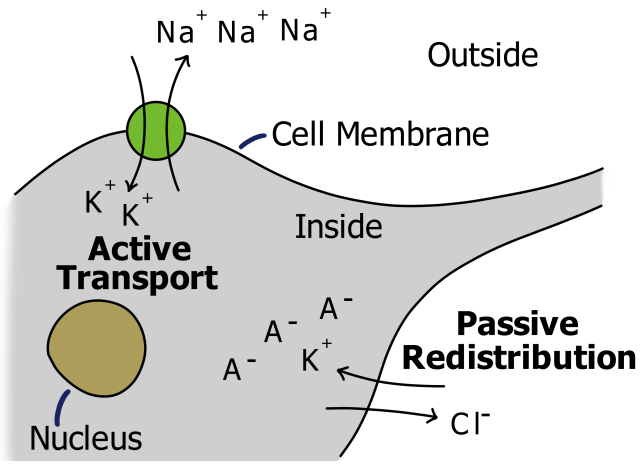


Figure 2.3: A cell maintains its concentration gradient through both active transport and passive redistribution.

These concentration gradients are formed and maintained by two main mechanisms, *active transport* and *passive redistribution*, as shown in Figure 2.3. With passive redistribution, negatively charged anions locked within the body of the neuron attract

the positively charged K^+ (potassium) ions from the surrounding medium into the cells body. At the same time, negatively charged Cl^- (chlorine) ions are repelled from the cell. As such, the ions diffuse across the cell membrane, increasing the cells potential relative to the surrounding medium. Active transport, on the other hand, makes use of ion pumps to drive ions against the concentration gradient. This process requires energy and allows the neuron to reach potentials outside of its resting equilibrium. By these two mechanisms, the neurons are brought to a relatively steady resting membrane potential of -70mV [12].

Ions travel across the cell membrane though *ionic channels*. Some of these channels include gating particles that open and close the channel based on some pre-defined external properties. Typical channel sensitivities include the membrane potential (termed *voltage-gated* channels), intracellular agents and extracellular agents (such as neurotransmitters and neuromodulators) [13].

These gates may be separated into two different groups: the *activation gates* and the *inactivation gates*. Activation gates open the channel when stimulated, while inactivation gates close the channel under stimulus. These two processes are totally independent of one another, allowing a channel to be both activated and inactivated at the same time.

Ionic currents may therefore be formed across the cell membrane by the opening and closing of different ionic channels. This process gives neurons their ability to generate APs in response to external stimuli, as described in the following Section.

2.2.2 The Action Potential

The generation and transmission of APs plays a critical role in a neurons function within the nervous system. The typical components that constitute both a Graded Potential (GP) and an AP signal are shown in Figure 2.4. GPs are characterised by their sub-threshold depolarization, leading to a brief repolarization period as the capacitive cell membrane potential restores to its resting potential. During this repolarization period, any additional sub-threshold input stimuli received may compound to move the membrane potential over the threshold, resulting in a full AP. In this manner, neurons operate as if they were leaky integrators, where incoming signals additively depolarise the cell membrane before slowly restoring to the resting potential. When, and only when, the cell membrane is depolarised beyond an internal threshold value the all-or-nothing AP is generated resulting in a pulse that travels down the cells axon.

The AP, as shown on the right of Figure 2.4, is characterised by its supra-threshold depolarisation followed by a relatively sizeable upstroke or 'spike'. The cell membrane then undergoes repolarization and shoots past the resting potential for a brief refractory period in what is known as *hyperpolarization*. During this refractory period excitatory

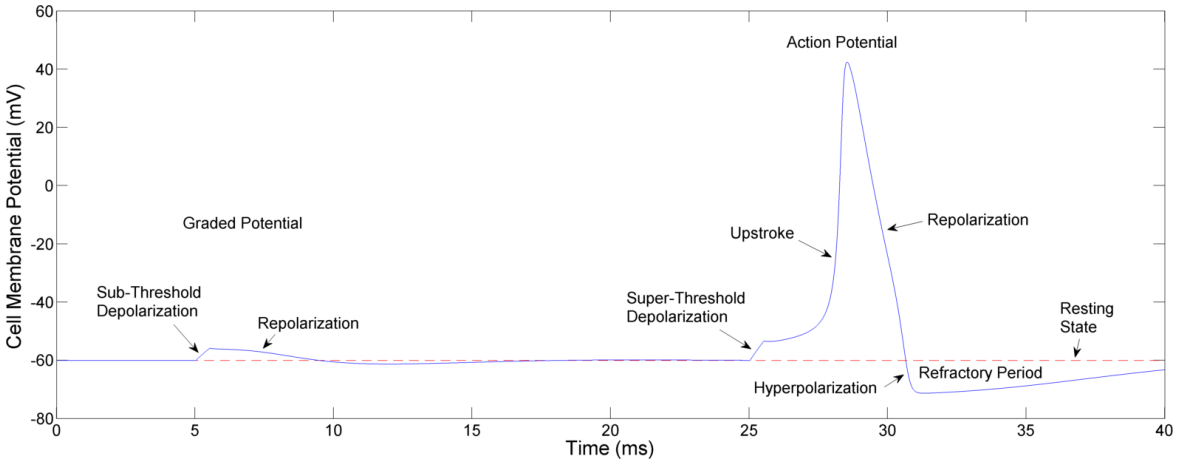


Figure 2.4: A cell membrane potential when undergoing a graded potential and action potential as defined by the traditional Hodgkin-Huxley neuron model described in Section 2.4. Figure adapted from *Dynamical Systems in Neuroscience*, by E. Izhikevich [13].

input stimuli must first overcome the hyperpolarization of the cell membrane before they may trigger another AP.

These two potentials begin at the soma, which receives a range of inputs from various sources via the dendrites. The received stimuli lead to a disturbance or depolarisation of the membrane potential at the axon hillock. The threshold of the neuron dictates the point at which the neuron undergoes a state change at the axon hillock, resulting in an inrush of Na^+ (sodium) ions. This inverts the cell membrane potential, leading to the large upstroke seen in APs. Following a short delay, the K^+ gated channels are also opened, allowing K^+ ions to flow out of the cell, resulting in repolarization. This is a local effect, however the localised positive charge generated during the upstroke triggers adjacent Na^+ and K^+ gated channels to open. In this way, the AP is propagated down the full length of the axon, with the charged ions moving perpendicularly to that of the AP itself, as demonstrated in Figure 2.5.

The stimulus required to trigger rapidly successive APs must first overcome the hyperpolarization generated by the AP during its refractory period. This hyperpolarization is important in ensuring that an AP does not trigger a continuously reflected potential back along the axon.

2.2.3 The Synapse

Once an AP is generated it travels down the axon to the *Telodendria* where it arrives at the pre-synaptic terminals. These pre-synaptic terminals connect and communicate with other neurons by means of a structure known as a *Synapse*. Synapses allow neurons to pass signals to connected cells and can operate as electrical or chemical connections. For chemical synapses, neurotransmitters released by the pre-synaptic

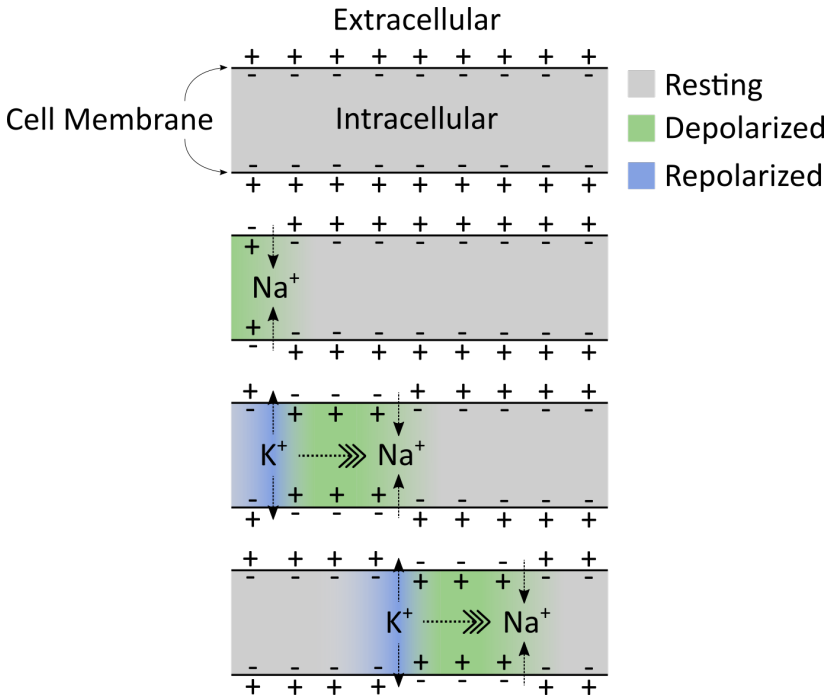


Figure 2.5: Nerve cut-through showing an AP travelling along an axon left-to-right. Ionic conduction is perpendicular to the APs direction of motion, resulting in rapid transportation of charge.

terminal bind to receptors in the post-synaptic terminal. These neurotransmitters may trigger an electrical response or a secondary messenger pathway that excites or inhibits the post-synaptic neuron. In electrical synapses a current is conducted by special channels termed *gap junctions*, exciting the post-synaptic neuron. These electrical synapses are typically faster than their chemical counterparts.

Synapses allow neurons to trigger or inhibit the production of APs in other neurons within the neural network. The degree to which a pre-synaptic neuron affects a post-synaptic neuron is determined by the synapse strength or weighting. This weighting is a tunable parameter and plays a critical role in the formation of memory and learning. The combination of synapse weighting and internal thresholds allows neural networks to perform complex data processing, in ways that can be compared to the application of simple logic gates in ICs. It is the decision making process behind the generation of APs that provides the computational power of neural networks, just as it is the decision making process behind simple logic gates that allows processors to process data.

Any one neuron can have thousands of synaptic connections resulting in densely connected networks with complex internal relationships. The degree of connectivity is largely dependent on the role of the neuron in question, with CNS neurons typically utilising more connections than their PNS counterparts.

2.3 Modelling Neurons

The actual mechanics behind the generation and transmission of APs involves complex dynamics. As a result, modelling these systems requires certain concessions or approximations. The level of approximation used in any neural model will directly affect its utility in neural and computational research, and must be decided with reference to the intended target application.

Biophysically accurate models are required when studying the underlying function of the biological nervous system. This leads to significant computational overhead and is largely due to the inherently complex relationships between the many biophysical mechanics of an individual neuron. As the relevance of each feature of a neurons function is currently under investigation, it is not possible to disregard the many small facets of neuron dynamics without compromising the integrity of the experimental results. These models, known as kinetic models, will often reflect the biological function of the neuron at an ionic level, simulating the kinematics of multiple different ion populations within the neuron. There are a large number of these conductance based models implemented in hardware [14]. Each uses electric current to simulate some combination of ion channels, as deemed significant by the models designers.

At the other extreme, computationally efficient models are required when simulating massive networks of neurons. These higher-level investigations are often more interested in the compound performance of the network as a system itself, and the large number of neurons used in these investigations makes biophysical accuracy impracticable. Such models will often focus on the AP itself and are therefore less concerned with the underlying cells kinematics. In these models the generation, timing, transmission and/or morphology of an AP is more critical, leading to a large number of simplified computational models.

Between these two extremes of modelling accuracy and efficiency exists a varied assortment of neuron models. Each provides some level of abstraction and biophysical accuracy as deemed necessary for a given application or research effort. This trade-off between computational efficiency and biophysical accuracy is visualised in Figure 2.6, showing a set of model classes and popular examples where appropriate. There are a small number of models that exist outside of this scale, such as the Izhekevich model, discussed later in Section 2.5.

Even in the design of computationally efficient models, a good understanding of neuron function is required to inform the selection of appropriate approximative models. As the field progresses, the requirement to produce more accurate and yet large scale simulations continues to drive the development of new neural models and systems.

The remainder of this chapter will provide an example of two different biophysically accurate models. Starting with the famous Hodgkin-Huxley model, Section 2.4 will introduce many key neural modelling concepts. The Izhekevich model will then be

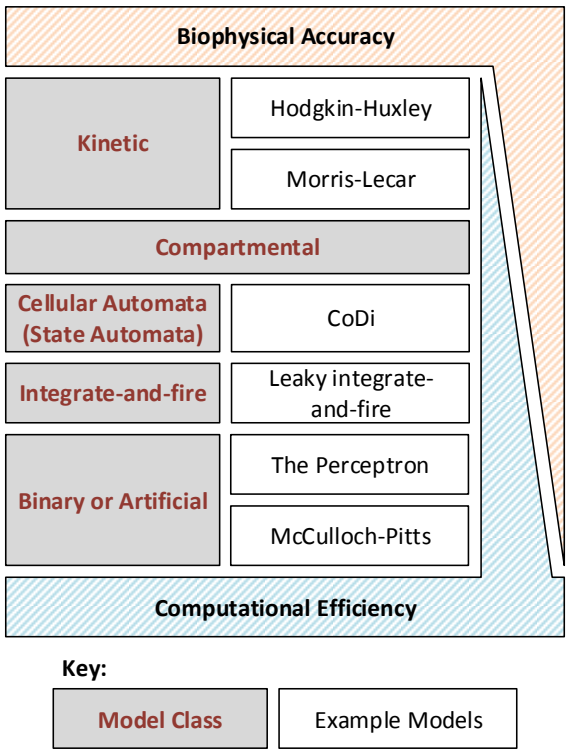


Figure 2.6: The trade-off between biophysical accuracy and computational efficiency for a number of classes of neuron models. Where appropriate, popular examples of each class are provided for reference. Figure adapted from *Towards the neurocomputer: An investigation of VHDL Neuron Models*, by J. Bailey [15].

introduced in Section 2.5, showing its ability to provide somewhat accurate representations of APs efficiently, at the cost of approximation elsewhere within the model. Following this focus on the biophysically accurate models, Chapter 3 then reviews computationally efficient models in greater depth.

2.4 The Hodgkin-Huxley Model

The Hodgkin-Huxley model is arguably one of the most influential conductance based models in computational neuroscience [16]. This model was developed by Alan L. Hodgkin and Andrew F. Huxley using experimental recordings of the current-voltage relationships seen in a squid (*Loligo forbesi*) giant axon [17]. Using these readings, Hodgkin and Huxley calculated the form of the recorded APs in a separately published analysis [18]. Both Alan Hodgkin and Andrew Huxley were awarded the Nobel Prize alongside Sir John Carew Eccles in 1963 for "their discoveries concerning the ionic mechanisms involved in excitation and inhibition in the peripheral and central portions of the nerve cell membrane."

Through these experiments, it was shown that the majority of the recorded membrane current was due to K^+ and Na^+ ion conductance. For this reason, the Hodgkin-Huxley model focuses on these two ionic channels. While other ionic currents exist within neurons, it is commonly assumed that these currents are negligible and may be safely approximated using an additional leakage current within the model. The membrane capacity was found to posses similarities to that of a perfect condenser, and it was therefore modelled using a capacitor. Building upon these findings, Hodgkin and Huxley defined the total current density, I, which represents the net current flowing across the cell membrane due to all ionic conductance, as follows:

$$I = C_M \frac{dV}{dt} + I_{Na} + I_K + I_l \tag{2.1}$$

where I_{Na} is the Na^+ current density, I_K is the K^+ current density, I_l is the leakage current, C_M is the membrane capacity per unit area and V is the membrane potential.

The Equivalent Circuit Model

An equivalent electrical circuit, shown in Figure 2.7, was developed by modelling the gated ion channels as variable resistors [18]. Since the ionic currents are independent to one another, the ion channels (labelled 'Active' in Figure 2.7) are each treated as isolated current paths. The capacitance and diffusive leakage of the membrane were also included as two additional current paths. Extra ionic currents may be included in this model through the addition of more active channels, however the leakage conductance, g_l, must be recalculated to account for such modification.

The different levels of charged ions on either side of the membrane generates a potential across the membrane, termed the *Nernst potential*. In the equivalent circuit, the potential generated by each ion population is shown as a unique voltage source for the related conductance channel. The Nernst equilibrium potentials for the Na^+, K^+ and leakage ions used in the original model are listed in Figure 2.7.

Channel Gating Probabilities

The rate at which Na^+ and K^+ ions flow through the cell membrane (that is, I_{Na} and I_K) is controlled by the Na^+ and K^+ channel conductances, g_{Na} and g_k. Both of these channels are dynamic, resulting in conductances that change with respect to the proportions of opened ion gates within the membrane. The probability of a K^+ activation gate being in the open state, n, may be defined according to the activation

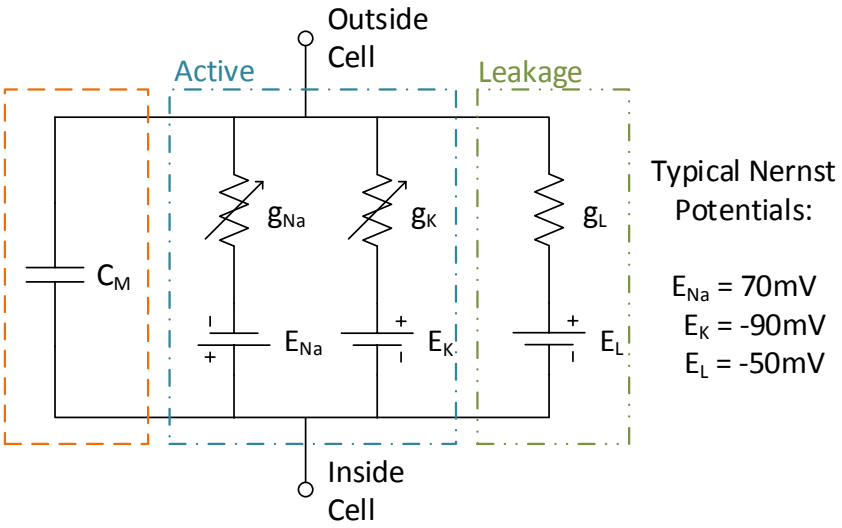


Figure 2.7: Equivalent circuit for the Hodgkin-Huxley model of an excitable cell membrane showing the active ion channels, leakage current and membrane capacitance, with the physical parameters used in Equation 2.10 also identified. Figure adapted from Hodgkin and Huxley's original paper [19].

rate, α_n, and deactivation rate, β_n, as follows:

$$\underset{\text{Deactivated}}{1-n} \underset{\beta_n}{\overset{\alpha_n}{\rightleftharpoons}} \underset{\text{Activated}}{n} \tag{2.2}$$

While K^+ is controlled by activation gates, Na^+ is controlled by both activation and inactivation gates. Na^+ ions are therefore unable to pass through the channel when it is inactivated, even if the activation gate is open. The probabilities that a Na^+ activation gate is open, m, may be defined using the activation and deactivation rates as before.

$$\underset{\text{Deactivated}}{1-m} \underset{\beta_m}{\overset{\alpha_m}{\rightleftharpoons}} \underset{\text{Activated}}{m} \tag{2.3}$$

The probability that a Na^+ inactivation gate is open, h, is defined in the same fashion. It is common practice to say that open inactivation gates are *deinactivated*.

$$\underset{\text{Inactivated}}{1-h} \underset{\beta_h}{\overset{\alpha_h}{\rightleftharpoons}} \underset{\text{Deinactivated}}{h} \tag{2.4}$$

As with any rate controlled population distribution, these activation and inactivation probabilities may be redefined as a differential with respect to time. The method for this conversion is the same for all three channel probabilities and may found under

Appendix A. Applying this conversion to Equations 2.2, 2.3 and 2.4 yields the following results:

$$\frac{dn}{dt} = \alpha_n(1-n) - \beta_n n \tag{2.5}$$

$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m m \tag{2.6}$$

$$\frac{dh}{dt} = \alpha_h(1-h) - \beta_h h \tag{2.7}$$

Modelling the Activation Rates

Thus far, the activation and inactivation rates have been treated as steady state values. This is actually not the case for a neuron, where they may be shown to be dependant upon the cell membrane potential. The relationships between activation rates and the membrane potential were found and modelled by Hodgkin and Huxley, resulting in the equations shown in Table 2.2.

Table 2.2: Activation and inactivation rates for the K^+ (potassium) and Na^+ (sodium) channels of a squid giant axon.

K^+ Activation	Na^+ Activation	Na^+ Inactivation
$\alpha_n(V) = 0.01\frac{10-V}{\exp\left(\frac{10-V}{10}\right)-1}$	$\alpha_m(V) = 0.1\frac{25-V}{\exp\left(\frac{25-V}{10}\right)-1}$	$\alpha_h(V) = 0.07\exp\left(\frac{-V}{20}\right)$
$\beta_n(V) = 0.125\exp\left(\frac{-V}{80}\right)$	$\beta_m(V) = 4\exp\left(\frac{-V}{18}\right)$	$\beta_h(V) = \frac{1}{\exp\left(\frac{30-V}{10}\right)+1}$

Calculating the Channel Conductances

With the activation rates defined, the number of open activation and inactivation gates may be calculated using equations 2.5, 2.6 and 2.7. Once the gating probabilities are known, the channel conductances may be found using the following equations.

$$g_K = \bar{g}_K \; n(V,t)^4 \tag{2.8}$$

$$g_{Na} = \bar{g}_{Na} \; m(V,t)^3 h(V,t) \tag{2.9}$$

In these equations, $\bar{g}$ represents the maximal conductance of the respective ion channels, defined as $\bar{g}_K = 36$ mS/cm^2, $\bar{g}_{Na} = 120$ mS/cm^2 and $\bar{g}_L = 0.3$ mS/cm^2. The powers used in these equations represent the ratios of activation and inactivation gates and their values were selected by Hodgkin and Huxley to provide the best fit for the recorded data from the squid giant axon.

Complete Model

By combining Equations 2.1, 2.8 and 2.9, along with general the relationship $I = gV$, the full Hodgkin-Huxley equations may be realised as follows:

$$C_M \frac{dV}{dt} = I - \overbrace{\bar{g}_K\ n^4(V - E_K)}^{I_k} - \overbrace{\bar{g}_{Na}\ m^3 h(V - E_{Na})}^{I_{Na}} - \overbrace{\bar{g}_L(V - E_L)}^{I_L} \tag{2.10}$$

$$\begin{aligned} \frac{dn}{dt} &= \alpha_n(V) \cdot (1 - n) - \beta_n(V) \cdot n \\ \frac{dm}{dt} &= \alpha_m(V) \cdot (1 - m) - \beta_m(V) \cdot m \\ \frac{dh}{dt} &= \alpha_h(V) \cdot (1 - h) - \beta_h(V) \cdot h \end{aligned} \tag{2.11}$$

where I is the injected current.

These equations are often converted into a steady-state/time-constant form that then allows the system to be modelled using Euler's method of approximation. This conversion is demonstrated in Appendix B. The temporal membrane potential response to any stimulus may then be calculated using this model. This models response to a 0.2mA pulse is shown in Figure 2.8 alongside the internal gating probabilities.

2.4.1 Summary

Since the original publication, this model has been modified in many different ways to include extra dynamics or ionic channels. Despite these modifications, the original model continues to be regarded as the "golden model" of neuron dynamics. It is commonly used as reference when designing new biophysically accurate or approximative models. The popularity of this model is largely due to the rigorous biological recordings used in its generation. While this offers a good reference for model design it should therefore be noted that the model is based on measurements of a squid giant axon and may require further tuning for applications involving other biological nervous systems. With differences in operational temperature and physical scale, an accurate model of a human

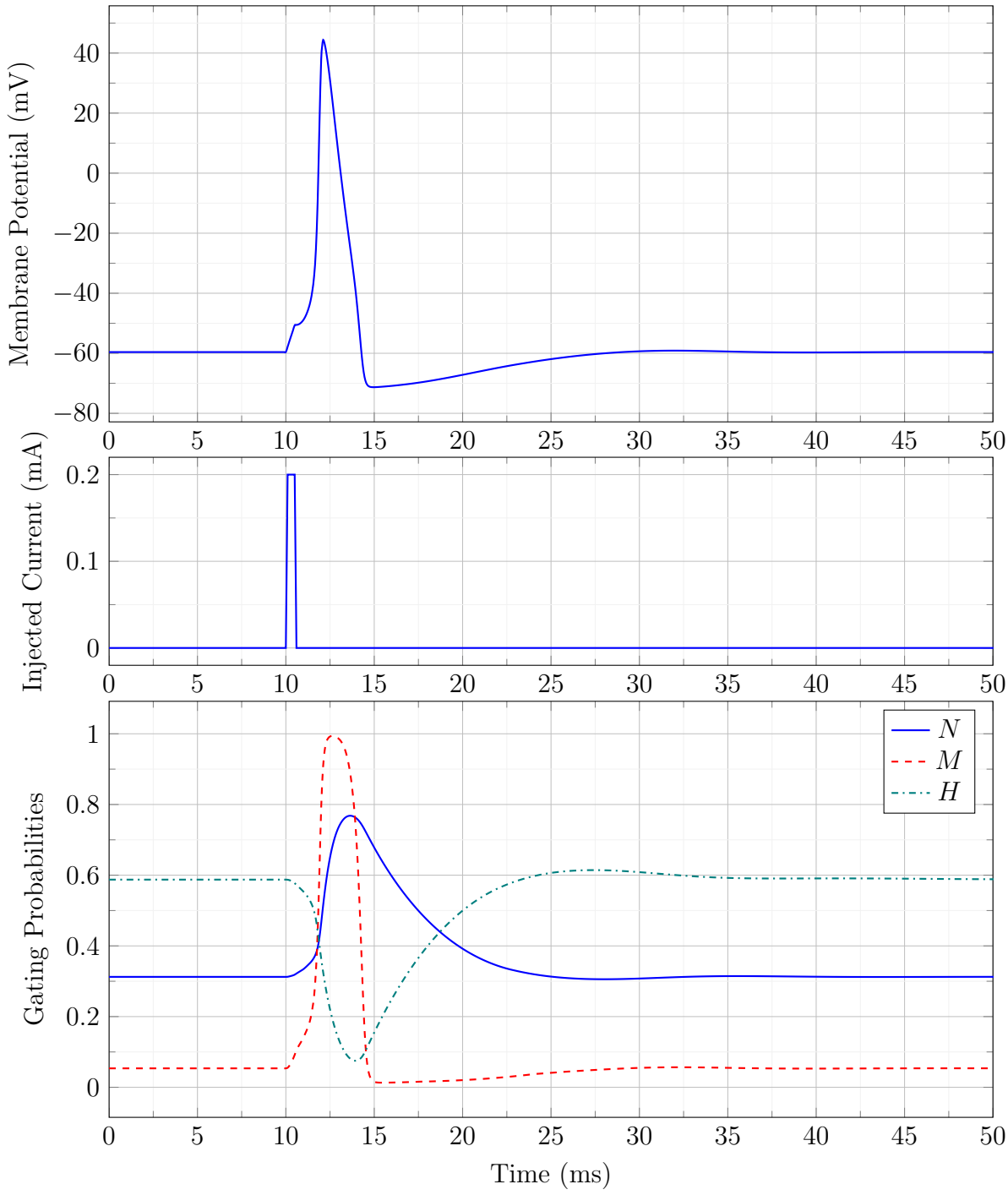


Figure 2.8: Hodgkin-Huxley model membrane potential response to an injected pulse input of 0.2mA, showing the probabilities of ionic gates being in the open position. This figure shows the characteristic AP shape previously discussed in this chapter. These waveforms were generated using a discrete-time MATLAB simulation.

neuron may require a significant change in the underlying model parameters to that of a squid giant axon.

Partly due to its biophysical complexity, the Hodgkin-Huxley model is very computationally expensive, requiring the solution of an Ordinary Differential Equation (ODE) for each of the ionic gating probabilities shown in Equation Set 2.11. An additional ODE, shown in Equation 2.10, must then also be computed using these new values, resulting in a time consuming calculation that must be performed for each and every simulated time step. As a result of this high computational cost, it is very uncommon to see the Hodgkin-Huxley model used in simulations of more than a few hundred neurons.

2.5 The Izhikevich Model

The Izhikevich model was first developed by Eugene Izhikevich in an effort to provide an efficient and yet biophysically accurate model for neuroscience and neuromorphic research [20]. Unlike the Hodgkin-Huxley model, which started by considering the underlying mechanics of a neuron, Izhikevich began by instead focusing on the APs properties and raw shape. When considering other spiking models, it was found that many of them could be reduced to two-dimensional systems consisting of a fast voltage variable and a slower recovery variable, used to describe some combination of activation of the K^+ current or inactivation of the Na^+ current [13]. The Izhikevich model was therefore designed to reflect this pattern, building upon the assumption that the AP morphology itself is less important than the sub-threshold dynamics leading to the AP in question. As such, this model retains the detailed dynamics about the resting potential and threshold range, using a simplified vector field model once operating outside the threshold neighbourhood.

Building upon these assumptions and observations, the Izhikevich model achieves a close representation of neural function while maintaining a deceptively simple mathematical format. This model may be represented as follows:

$$C\frac{dv}{dt} = k\left(v - v_r\right)\left(v - v_t\right) - u + I \tag{2.12}$$

$$\frac{du}{dt} = a\left\{b\left(v - v_r\right) - u\right\} \tag{2.13}$$

$$\text{if } v \geq p \text{ then } v \leftarrow c,\ u \leftarrow u + d \tag{2.14}$$

In this form, v is the membrane potential, u is the recovery current, C is the membrane capacitance, I is the input current, v_r is the membrane resting potential and v_t is the

threshold potential. Both a and b are scalar parameters used to control the steady-state sub-threshold behaviour. Through careful tuning, these parameters may represent either an integrator or resonator model. Parameters c and d may also be used to modify the post-spike transient behaviours of the model.

The parameter a may be associated with the recovery time constant. The sign of the parameter b determines whether the recovery current is an amplifying ($b < 0$) or resonant ($b > 0$) variable. If $b \leq 0$ the model may be considered a quadratic integrate-and-fire neuron, with non-zero values providing an additional passive dendritic compartment resulting in the attenuation of the signal in accordance with passive dendrite models. When operating in resonant mode, the model represents a new class of spiking neural models as defined by Izhikevich [13]. Finally, the reset parameter, c, determines the magnitude of the membranes reset voltage, while the peak voltage threshold, p, determines the reset condition.

2.5.1 Summary

Despite the simple form of this model, it has been found capable of representing all 20 of the fundamental spiking patterns identified by Izhikevich [21]. With an estimated compute requirement of around 13 FLOPS/ms, this model provides a highly flexible yet efficient implementation of spiking neural function. The potential performance gains found using this model are made clearer when compared against the Hodgkin-Huxley models estimated compute requirements of 1200 FLOPS/ms. Despite the 92× improvement in efficiency, it is important to note that Izhikevich identifies the model as unsuitable for generating biophysically accurate results. The Izhikevich model is better suited to computational neuroscience simulations and Artificial Neural Network (ANN) implementations. Providing a practical solution so long as the approximated non-threshold-neighbourhood AP dynamics of the model are acceptable for the intended application.

2.6 Simulating Neuron Models

Sections 2.4 and 2.5 have introduced two different neural models, both are used commonly in research today. While the Izhikevich model offers greater computational efficiency than that of the Hodgkin-Huxley model, its loss of biophysical accuracy makes it unsuitable for some research. In practice, there are a large number of neuron models, each designed for its own specific qualities and features. Developing robust simulations of each of these models represents a significant time investment. As a result, pre-made software simulators have become a core staple of neural research - reducing the development time while also providing a verified and widely tested simulation engine upon which meaningful results may be produced. The wide availability of these

implemented models also makes the validation and comparison of findings a practical task, improving the scientific rigour with which neural research may be performed.

There are many neuro-simulation tools and resources available, providing a range of theoretical and empirical data and encouraging the study of both system-wide and small-scale neuronal behaviour [22]. Selection of the appropriate tool or resource often depends on the application or mechanism under investigation. Each of the simulators provides different levels of complexity and scale, with some focused on the accurate modelling of a single neurons finer mechanics, while others provide larger systems of neurons better suited to investigations involving the connectivity and structure of the systems in question. This section provides a brief summary of three different simulation efforts, considering their intended applications.

2.6.1 NEURON

NEURON, developed at Yale University, is a simulation environment designed to support the investigation of both individual neuron models, as well as whole network implementations. This highly flexible tool offers an application domain that extends beyond the continuous simulation of complex anatomical and biophysical properties. With the appropriate settings, this system may also be used to explore both discrete-event models and hybrid simulations that combine biological and artificial neural models [23].

Through careful support of parallel processing, NEURON can achieve at-least-linear reductions in runtime requirements as the number of processors is increased [24]. The gains achieved through the addition of new processors was found to last until each processor is handling about 100 equations. NEURON supports Windows, Linux and MacOS making it easy to include in established research pipelines. With an active community of researchers, NEURON has been shown suitable for both teaching [25] and research roles [26].

Neuron is best suited for detailed model implementations, such as biophysically accurate neural models. In cases where large networks are required, however, there are a number of alternative simulators which perform such tasks in a more effective manner [27].

2.6.2 NEST

The NEural Simulation Technology (NEST) initiative represents a collaborative effort to develop a new neural software framework. The simulator is designed to simulate large scale, structured neuronal systems, supporting heterogeneous biophysically accurate elements at varying detail levels. In a manner similar to that of the Izhikevich model, this biophysical accuracy is only maintained in regions deemed as ‘points of interest’,

with the rest of the model abstracted to faster and more efficient approximative implementations [28].

Designed to utilise parallel computation, NESTs underlying model implementations have strict reproducibility requirements. This means that network performance is deterministic, regardless of the number of processors used during computation. Controlled using a high-level expressive language, neuron and synapse models must be implemented as C++ classes, providing a common pre-defined set of arguments to allow the end user to rapidly switch between different neural models within a given experiment [26, 28].

Distributed computing systems, such as those supported by NEST, are capable of easily simulating 10^5 neurons at speeds suitable for practical work [29]. This could easily make NEST a viable option alongside custom hardware systems when considering the acceleration of large neural simulations. Despite this promising result, the overheads involved in CPU based simulators make such competition unlikely, with the model flexibility offered by this simulator as the only real benefit over the custom hardware solutions that are typically more efficient and faster.

2.6.3 PyNN

Unlike NEURON and NEST, PyNN is a simulator-independent network modelling API developed to support and encourage collaborative work. Davison *et. al.* identify the many different simulators used by research teams as a significant barrier to sharing work and building upon the findings of others within the field of neural simulation [30]. Each simulator and tool requires the user to specify the problem or model in a unique way. With the rapid onset of changes and updates commonly associated with collaborative development projects, this difference in specification between models makes the verification and reproduction of findings a challenging and time consuming task.

PyNN was developed to address this issue, providing an abstracted simulation-independent front end. This means that users need only specify the network structure once, running the defined network on multiple simulators without changing the original system definition. This tool makes cross-validation of simulation results possible, ensuring that features of interest are indeed a result of neuronal function and not a simulation artefact.

2.6.4 Summary

Each of these simulators are readily available to the general public, with their development ongoing under open-source licences. While the open availability of these

systems should encourage further use and development, there is often a perceived barrier to entry. This is largely due to the complex and custom APIs that must first be studied prior to the implementation of any models. Such problems are not uncommon with large and powerful software tools, however the small research community makes the production of suitable guides and resources challenging. A problem unfortunately emphasised by the continued modification and incremental development cycles of open-source solutions.

PyNN represents a significant joint effort to address this issue, providing users with a singular problem specification syntax that supports cross-simulator implementation and verification. Such generalisation however comes at a cost, constraining new and novel models to fit within the specifications afforded by PyNNs syntax. Models existing outside of the scope of the PyNN API will therefore require additional features to be added to the PyNN system, adding further development and testing requirements to the new model under consideration.

Both of the simulators identified in this section support large and small scale parallel implementations of neural systems. This makes them especially suited for biophysically accurate models of networks of neurons. When considering applications where biophysical accuracy is not required, however, there are many hardware and software solutions that offer improved performance over these simulators. The implementation of computationally efficient neural networks is covered in greater detail in Chapters 3 and 4.

2.7 Conclusions

The understanding of biological nervous systems continues to drive developments within the neural network community. While many questions regarding the mechanics of thought remain unanswered, current research hopes to unlock the mysteries of the human brain through careful and precise modelling of its fundamental building blocks - such as the neuron. This chapter has outlined the operation of biological neurons, identifying the membrane characteristics critical to the generation of APs. In particular, the importance of gated ionic channels was shown, with the K^+ and Na^+ ion gradients playing a critical role in the generation of an APs characteristic shape. The synapse was considered and identified alongside the cells internal threshold as a key element in a neurons capability to process information.

The Hodgkin-Huxley model, made famous for its biophysically accurate representation of the neurons AP generation, identifies the membrane capacitance alongside the K^+ and Na^+ conductance as the three primary components of neural function. All other ions and electrical properties are combined into a singular 'leakage' conductance, resulting in a model that closely resembles results observed in a squid giant axon.

Widely recognised as the 'golden model' for neural AP generation, the Hodgkin-Huxley model is somewhat constrained by its high computational cost, requiring an estimated 1200 FLOPS/ms. In contrast other, more computationally efficient models often forfeit their biophysical accuracy in favour of improved scalability. Some solutions, such as the Izhikevich model, appear to exist outside of this accuracy-efficiency trade-off. This is achieved through careful selection of model regions that may be approximated without resulting in considerable accuracy loss. The Izhikevich model, for example, emphasises accuracy about the threshold and resting potential, yet yields a more approximative and efficient representation outside of this region. Applying this approximation rule to traditional neural modelling yields a model capable of fully representing spiking neuron behaviour while only requiring 13 FLOPS/ms. Use of such models, however, demands careful consideration regarding the effects of the approximations before results may be claimed biophysically relevant.

The selection of a suitable model for a neuron depends largely upon the end application. Biophysically accurate models are computationally expensive, making them unsuitable for mobile applications or low power devices. Equally, the computationally efficient solutions are often too approximative in their nature to yield meaningful results when studying biological neurons and their operation. There are a range of software solutions for implementing biologically accurate models of neurons, with both NEURON and NEST capable of simulating from fine-detail single models to large networks of neurons. The PyNN API offers researchers an abstracted simulator-independent implementation, allowing one model definition to be tested against multiple simulators without any requirement for re-definition. It is the general hope that these developments will lead to a better understanding of the functions and mechanisms behind the biological nervous system.

Biological neural networks will continue to drive innovation in a large number of fields, making them a critical subject when investigating ANNs and novel processor architecture design. As Moore's law reaches its inevitable limits, the inspiration afforded by our own nervous system may lead to critical developments in signal processing solutions. Despite the lack of biophysical accuracy, many computational neuron models have achieved state-of-the-art results in neural network research. Chapter 3 reviews the development and implementation of such models, which can often provide useful insight into the operation of the nervous system as a whole. Their relative simplicity allows investigations into large-scale systems of several million neurons and synapses; a scale that would be impractical when using models such as that of Hodgkin-Huxley.

Chapter 3

Artificial Neural Networks

The field of neural networks has received significant media attention, with reports of neural networks dreaming [31] and self-taught artificial Go champions [32] raising the public profile of artificial intelligence. This recent growth in public attention reflects the considerable growth seen in the academic field [33], the resurgence of which is likely a result of both the huge datasets now available for training these systems; and also the considerable growth in available and reasonably priced computing power.

Neural networks have continued to increase in both size and complexity as computers have grown in computational power, with the larger networks reaching sizes of hundreds of thousands of neurons and millions of synapses [34]. New structures have also been defined, allowing networks to parse a problem space in novel ways. As a result, neural networks have shown significant improvements in their abilities to process and classify data.

Chapter 2 has reviewed a number of biophysical neuron models, identifying how computational complexity and biophysical accuracy are weighed against one another, often depending on the end application of the system in question. Biological neurons, however, are not independent computational entities and instead form networks that have a processing potential that goes far beyond that of its constituent parts. The training and structure of these neural networks has a significant impact on the performance of the overall system. In this chapter three computationally efficient neuron models are described in Section 3.2. Commonly used activation functions are then considered in further detail in Section 3.3, before some common network structures are discussed in Section 3.4. Finally, the back-propagation algorithm is reviewed alongside other training methods in Section 3.5.

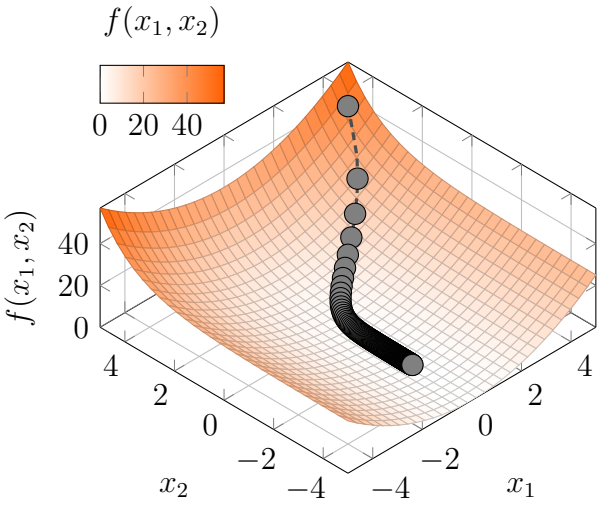


Figure 3.1: Gradient descent demonstration for the function $f(x_1, x_2) = {x_1}^2 + 2^{x_2}$. The dashed line shows the path that the algorithm follows, while the circles show the iterative values generated by each iteration of the algorithm. The algorithm starts at position [4.5, 4.5] and moves down the surface toward the global minimum.

3.0.1 Gradient Descent Overview

Before discussing and comparing Artificial Neural Networks (ANNs) it is important to first have a general understanding of how such systems may be trained. Gradient descent is a first-order iterative optimisation algorithm used to find the minimum of a function and forms a common tool used in neural network training. For this reason a high-level summary of the algorithm is provided here. For further information on the application of gradient descent algorithms in ANN training the reader is advised to review Section 3.5.

Gradient descent algorithms minimise functions by iteratively following the gradient of the function as shown in Figure 3.1. In many ways it is analogous to that of a ball rolling down a surface. In the case of neural network training, a gradient descent algorithm may be applied to the networks error function, resulting in iterative improvements in network performance. In order to achieve this, the differentials of the underlying network elements must be clearly defined (demonstrated in Equations 3.38 and 3.39 of Section 3.5). The 1st-order derivatives, and sometimes even 2nd-order derivatives of the neuron activation functions are therefore critical when applying a gradient descent training algorithm to an ANN.

Figure 3.1 also shows how small gradients can cause stagnation in gradient descent training procedures, with the amount that each iteration changes scaling proportionally with the underlying gradient geometry. As a result of this property, the iterations shown in Figure 3.1 are seen to slow as the optimisation progresses towards the flatter region of the problem space. This demonstrates a problem knows as the vanishing gradient problem, where a very-small gradient (such as those found on approach of

asymptotes) can trap the system in a sub-optimal region.

3.1 Natural and Artificial Neural Networks

The biological neuron has been described in Chapter 2. These neurons do not process data in isolation and are, instead, arranged into a large self-organising structure, termed a *neural network*. These neural networks are highly-parallel, performing complex data processing that far exceeds the capabilities of an individual neuron. The specific arrangement and layout of these networks is largely dependant on the tasks or role of the network in question. The Peripheral Nervous System (PNS), for example, is relatively sparsely connected, while the Central Nervous Systems (CNS) contains regions of densely connected neurons critical for the formation of higher consciousness.

Inspired by the nervous system, the term Artificial Neural Network (ANN) is used to describe systems comprised of simplistic computational units connected together through weighted connections. These systems typically utilise highly efficient but approximate neuron models. The connection weights dictate how the neural network operates as a whole and are usually chosen through some optimisation technique termed *training*.

The field of ANNs encompasses a wide range of structures and network classes and a set of the most common structures are discussed in Section 3.4 of this chapter. The neuron models used in ANNs are typically computationally efficient or approximative models to support large network implementations. These models are therefore described in greater detail below.

3.2 Artificial Neuron Models

Even with the considerable improvements in computational technologies, the task of simulating bio-physically accurate networks of more than several hundred neurons is currently impractical. As such, computationally efficient neuron models are often used in large to massive scale networks. These neurons models typically include some non-linear element and it is this non-linearity that provides the computational potential when these neurons are connected together in structured layers [35]. Without this non-linearity, any layers added to the network will simply modify the linear function already defined, adding no new degrees of freedom to the model.

3.2.1 The Binary Model

The first, and arguably simplest computational model of the neuron was the Binary or *McCulloch and Pitts* model, originally developed in 1943 by McCulloch and Pitts [36]. In this model the neurons have a set of one or more excitatory and inhibitory binary inputs and a singular binary output. These inputs are weighted with binary weights that take the form $w_i \in \{0, 1\}$ (i.e. enabled or disabled). The weighted inputs are summed together and this sum is compared against an internal threshold value to determine whether the output should be logical high or logical low. The fundamental binary nature of this model makes these basic neurons straightforward to implement in digital systems.

The inputs may be grouped together to apply varied weights to the incoming signals. For example, if a weight of 3 must be applied to an input signal 'A', one must simply connect signal 'A' to three separate input points, each with its own weighting of 1. Figure 3.2 shows the construction of this neuron, with n excitatory inputs, one inhibitory input and a step-threshold to condition the output.

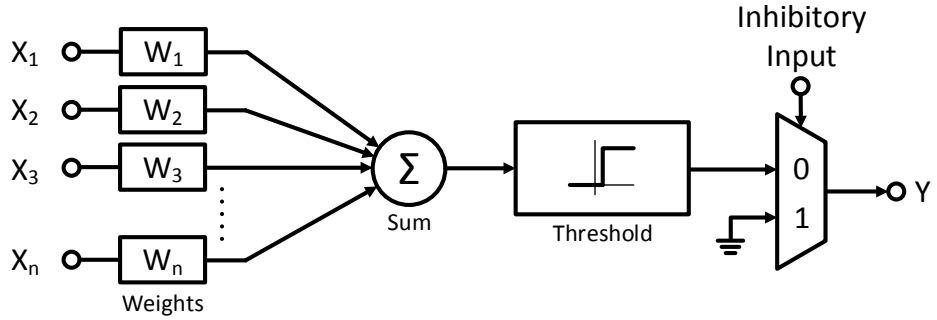


Figure 3.2: The binary model, showing the characteristic weighted inputs, internal threshold, absolute inhibitory input and multiplexed output. The binary nature of this model makes it especially suited for direct hardware implementation.

The model uses absolute inhibitory inputs, meaning that any high inhibitory signal will always produce a logic-low output regardless of any excitatory input values. Taking these properties into account, the McCulloch and Pitts model may be mathematically described as follows:

$$Y = \begin{cases} 1, & \text{If } \sum_{i=0}^{i=n} (W_i X_i) > T \text{ AND } \overline{\text{Inhibited}} \\ 0, & \text{Otherwise} \end{cases} \tag{3.1}$$

where Y is the output of a neuron, W_i and X_i are the i-th weight and input respectively, n is the total number of excitatory inputs and T is the threshold.

Using layers of neurons this model is capable of emulating any binary function, meaning that networks constructed from this model may be translated into a binary equivalent circuit of logic gates. Despite being pioneering work at its time, this model has seen little use within recent research efforts. In 1949, *The Organization of Behaviour* by Donald Hebb was published [37]. In this book, Hebb proposed a new rule for learning. Known as Hebb's rule, the assumption behind this rule is as follows:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." - *The Organization of Behaviour, Pg. 62* [37]

This rule means that the weights of the neuron must be flexible, supporting modification as a direct result of activation. The binary model does not support such modification by virtue of locking its weights at 0 or 1.

3.2.2 The Perceptron

The lack of weight modification support in the binary model was addressed by the 'Perceptron' model, developed in 1958 by Rosenblatt [38]. In this model the weights are real values, using positive values for excitatory synapses and negative values for inhibitory synapses, such that $W_i \in \Re$. Unlike the binary model, this model uses relative inhibition, meaning that the inhibitory signals may be overridden by any sufficiently weighted excitation. The use of real value weights means that the Perceptron model directly supports Hebbian learning as the weights may be adjusted incrementally instead of representing all or nothing connections. As shown in Figure 3.3, the signed weights mean that all inputs are treated equally - no longer requiring the isolation of inhibitory inputs. This means that inputs can become inhibitory during training without dependency on the pre-specification of input types.

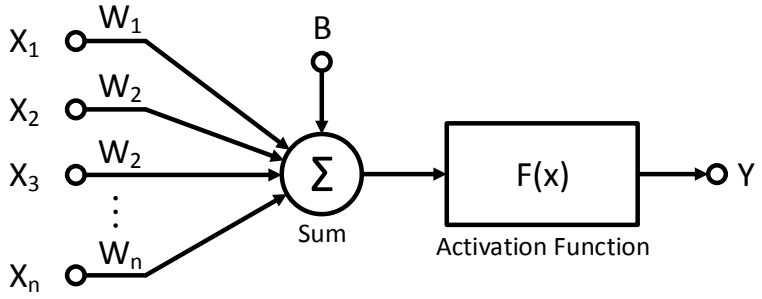


Figure 3.3: The Perceptron neuron model developed by Rosenblatt [38]. Unlike its binary model predecessor, this model uses real values and, typically, a non-linear activation function. An additional offset bias, B, allows the operational region of the activation function to be selected. These factors allow this model to represent arbitrary non-linear functions when arranged in a network.

The use of real values means that more complicated activation functions may also be used to shape the output of each neuron. Represented as $F(x)$ in Figure 3.3, these activation functions can range from simple step functions to complex polynomials and other non-linear equations. Commonly used activations functions are discussed in greater depth in Section 3.3 of this chapter.

A bias was also added to the Perceptron model, as shown in Figure 3.3. This bias allows the neuron to offset its default value, allowing the operational region of the activation function to be selected. In some implementations of the Perceptron model, the bias is replaced with a fixed input value of $X_1 = 1$ and a weight of $W_1 = B$. In this way the bias may be implemented without the requirement for an additional and task specific bias input. In either case, the underlying model may therefore be mathematically represented as follows:

$$Y = F\left(\sum_{i=0}^{i=n} (W_i X_i) + B\right) \tag{3.2}$$

where Y is the output, W_i and X_i are the i-th weight and input respectively, n is the total number of inputs (both excitatory and inhibitory), B is the bias and $F(x)$ is the activation function.

3.2.3 The Integrate and Fire Model

Both the binary and perception models represent highly simplified digital discrete neuron models, well suited for implementation on digital systems such as processors. While still highly approximative, the Integrate and Fire (IF) model instead reflects the analogue and temporal nature of neurons, making it a practical model that has found wide application in the design of artificial neurons. Originally developed in 1907 by Lapicque, this model accurately encapsulates the capacitance and leakage resistance of the cell membrane [39].

Figure 3.4 shows the original IF model, with the voltage, V_M, across a capacitor, C_M, representing the membrane potential; a resistor, R_{Leak}, setting the membrane leakage current; and a spike generation block. In this model it is assumed that the spike generator will produce an Action Potential (AP) when some internal threshold voltage is reached. This spike generation block also discharges the capacitor in the event of an AP, resetting the membrane potential to the resting potential, V_{Rest}.

Unlike the other approaches described in this chapter, the IF model is a conductance model, meaning that it simulates the flow of charged ions within the cell membrane directly using electrical currents within the circuit. Excitatory inputs are represented by sourcing current while inhibitory inputs are represented by sinking current. These currents drive charge onto the capacitor modifying the membrane potential. The

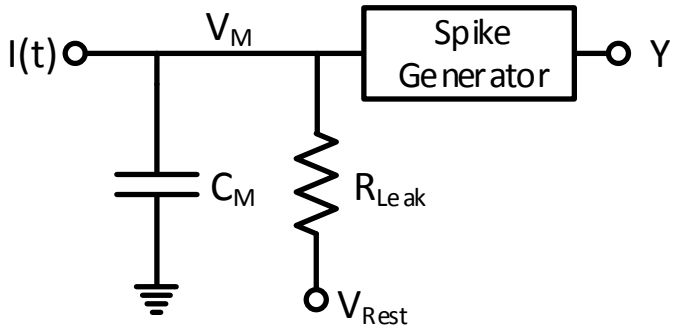


Figure 3.4: The integrate and fire model developed by Lapicque [40, 39]. In this model the resistor represents the membrane leakage current and the capacitor reflects the capacitive nature of the axon hillock in a biological neuron, resulting in a membrane potential. The spike generator block may be as complicated as desired and often depends on the intended purpose of the system itself.

leakage resistor allows the system to discharge over time, restoring the membrane to the default resting potential, defined by V_{Rest}, in the absence of stimulus. The refractory period may also be included in IF models by configuring the spike generator to return the membrane potential to a value below that of the resting potential whenever it is triggered. This hyper-polarization provides a short period of time where the neuron requires greater stimulus before it can trigger a secondary AP.

Both the binary and Perceptron model perform discrete summation of the incoming signals. In contrast the IF model uses integration to calculate the current stimulus level, so does not require the stimulus to arrive concurrently and, instead, supports the detection of any APs with suitably small temporal spacing.

3.3 Activation Functions

Activation functions form the final part of many neuron models. They shape the output of the neuron, playing a critical role in the networks ability to represent a function or map a problem space. These functions typically include the non-linearities required for neural networks to perform an arbitrary mapping of input data to output signals. Alongside non-linearity, a number of different criteria are used to compare and assess different activation functions. These criteria are in no way requirements and instead have an impact on how the activation function affects training methods and general network operation.

Order of Continuity A function is said to be *continuous* if its gradient is defined and non-infinite for all input values. The order of continuity is a measure of the differential continuity of a function. Notated as C^n, where n is the order, this measure yields the number of times that a function may be differentiated before

it becomes discontinuous. Discontinuous functions, that is functions that contain regions where the gradient is undefined, have an order of continuity of C^{-1}. Such functions do not support network training algorithms where the differential is required to inform the training process, since the undefined gradient regions (at the points of discontinuity) can cause the system to become trapped in, or oscillate around, a non-optimal solution. Continuous functions with discontinuous first-order derivatives (C^0) can be used, however mitigation must be performed for the lack of derivative continuity while training. The 1st- and 2nd-order differentials are often used in advanced training algorithms. As a result, the order of continuity is an important measure when considering new activation functions.

Range The range of a function impacts the amount by which weights will be modified during training. In practice it only really matters whether a function is finite or infinite since any finite function may be scaled to fit a desired range. Finite functions are typically more stable during training, however at the limits of such functions the gradients are often very small. In gradient descent training algorithms this can result in the vanishing gradient problem, where these small gradients cause the weights to become stuck regardless of input values.

Approximates the Identity Function Functions which approximate the identity function for small input values can learn efficiently when initialised with small random values.

Monotonic A monotonic function is one where the gradient is either all positive or all negative, with gradients of zero allowed in either case. Such functions are popular in gradient decent algorithms as they do not have local maxima or minima that can otherwise impede the traversal of the problem space. Networks using such functions can utilise simpler training algorithms, making them a popular choice in neural network implementations.

Asymptotic Asymptotic functions have regions where the output tends towards a fixed value as the input reaches some limit. Functions with asymptotic extremities, such that their output approaches a steady state value as their input approaches infinity, map an infinite input range onto a finite output range. The gradient of these functions becomes infinitesimally small for very large input values, however it crucially never reaches zero since the asymptote itself is never reached. This means that gradient descent algorithms may still be applied without risk of the system becoming trapped indefinitely at the functions extremities.

While the necessity for each of these criteria is somewhat subjective, they are useful when considering the impact of applying new activation functions to a network. There are a handful of functions that have seen popular use in neural networks. The most common of these are now introduced in Sections 3.3.1 to 3.3.5.

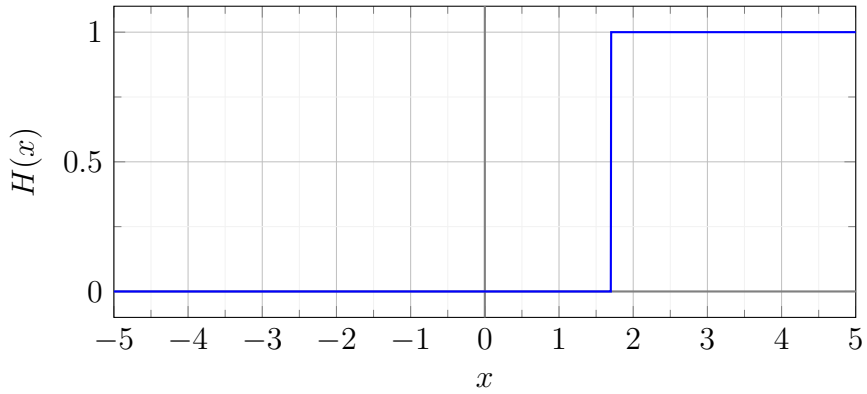


Figure 3.5: The binary step or threshold function, with an example threshold of $T = 1.7$. While this function traditionally has an output $H(x) \in [0, 1]$, some models use scaled or shifted binary step functions.

3.3.1 Binary Step

The binary step function is arguably the simplest non-linear activation function. Used in some of the earliest neural networks, this function acts as a basic thresholding operation.

Shown in Figure 3.5, this function may be described mathematically with a threshold, T, as follows:

$$H(x) = \begin{cases} 0, & x \leq T \\ 1, & x > T \end{cases} \tag{3.3}$$

The threshold value may also be treated as an additional weighted input. In such instances this extra input is added to the neuron with a fixed value of 1. A weight, w_0, is applied to this fixed input and scaled to ensure that $w_0 = -T$. This restructure ensures that the decision region is locked at zero, yielding the following form:

$$H(x) = \begin{cases} 0, & x + w_0 \leq 0 \\ 1, & x + w_0 > 0 \end{cases} \tag{3.4}$$

This function is easy to implement in digital systems due to its binary nature and formed one of the key components in the McCulloch and Pitts neuron models [36]. Such simplicity, however, comes at significant cost. Unlike many of the more complex activation functions, the binary step function is both non-continuous and non-differentiable in the threshold region. Alongside this issue, the gradient for this function is also zero for all other values. These limitations combined make the binary step function unsuitable for gradient descent training algorithms. The all-or-nothing

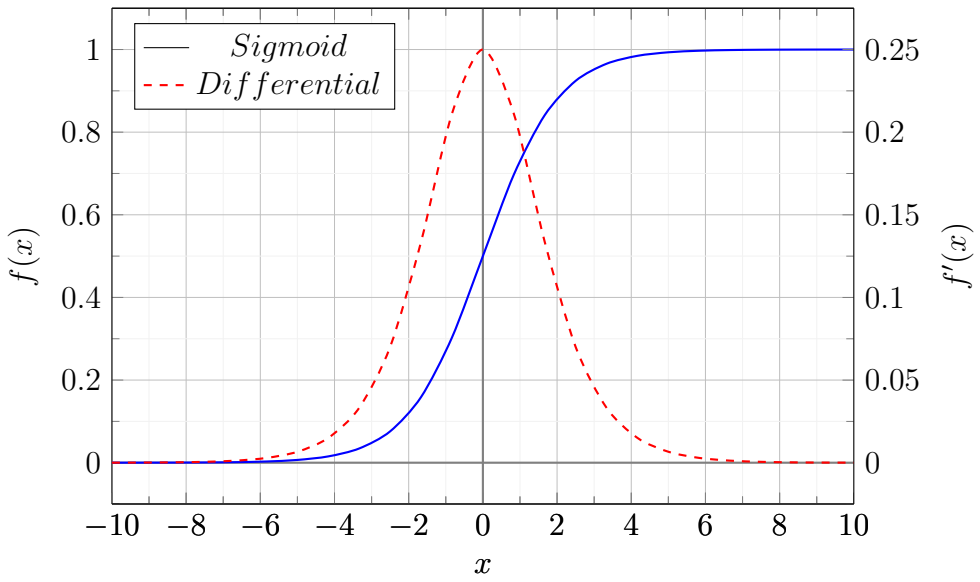


Figure 3.6: The logistic sigmoid function (solid blue) and its derivative (dashed red). The continuous and easily found differential makes this function a popular choice for neural networks undergoing gradient descent training.

nature of this function also means that no measure of error or certainty may be provided by the neurons, making it challenging to apply other training methods.

The binary step function is sometimes modified to produce outputs where $H(x) \in [-1, 1]$. This modified function is known as the Bipolar Step Function.

3.3.2 Logistic Sigmoid

Unlike the binary step function, logistic activation functions have an infinite order of continuity (C^∞) and a differential of one for small input values, making them well suited for gradient descent training algorithms. Finding considerable use in neural network implementations, these functions are monotonic, asymptotic and continuous. They are also known as 'squashing functions' as they map the entire real axis into a finite output range.

Logistic functions are often recognisable by the characteristic S-shaped curve seen in Figure 3.6. These functions have the general form:

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}} \tag{3.5}$$

Where L is the curve's maximum value, x_0 is the x-value of the curve's midpoint and k is the curve's steepness.

The logistic sigmoid function is by far the most popular logistic activation function and is defined as follows:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.6}$$

The justification for this common use may be found when considering a probabilistic view of classification tasks [41]. Assuming that there are two classes C_1 and C_2, the probability that some entry, x, falls into one of these classes may be found using Bayesian statistics. Defining $p(C_k)$ as the class priors and $p(x|C_k)$ as the class-conditional densities, Bayes' theorem states that the posterior probability (that is, the probability that the entry x belongs to class k given the value of x) may be defined as follows:

$$p(C_1|x) = \frac{p(x|C_1)p(C_1)}{p(x|C_1)p(C_1) + p(x|C_2)p(C_2)} \tag{3.7}$$

This probability may be rephrased by first defining an intermediate function, a, such that:

$$a = ln\left(\frac{p(x|C_1)p(C_1)}{p(x|C_2)p(C_2)}\right) \tag{3.8}$$

Substituting Equation 3.8 into Equation 3.7 yields the following simplified form:

$$p(C_1|x) = \frac{1}{1 + e^{-a}} \tag{3.9}$$

The resulting posterior probability equation (Equation 3.9) is of the same form as that of the logistic sigmoid function shown in Equation 3.6. This relationship may also be seen graphically by plotting two overlapping class probabilities, with each represented by a normal distribution. Figure 3.7 shows such an arrangement, with the two classes located such that there is an overlap in the class regions at the origin. For any entry where $x = 0$ it may be seen that there is an equal probability that the entry will belong to either of the classes. At the extremities the probability that an entry belongs to a given class approaches either zero or one asymptotically. Plotting the probability that a given entry x belongs to class C_2 yields the characteristic S-curve of logistic functions, as shown in Figure 3.7.

Besides describing the underlying shape of a probabilistic classification surface, the logistic sigmoid function also reflects the learning rate seen in success-based learning trials. Leibowitz *et. al.* demonstrated that the skill improvement seen when monitoring the effects of learning closely resembles that of a logistic function [42]. It therefore seems likely that the logistic function plays a role in natural learning, just as gradient descent

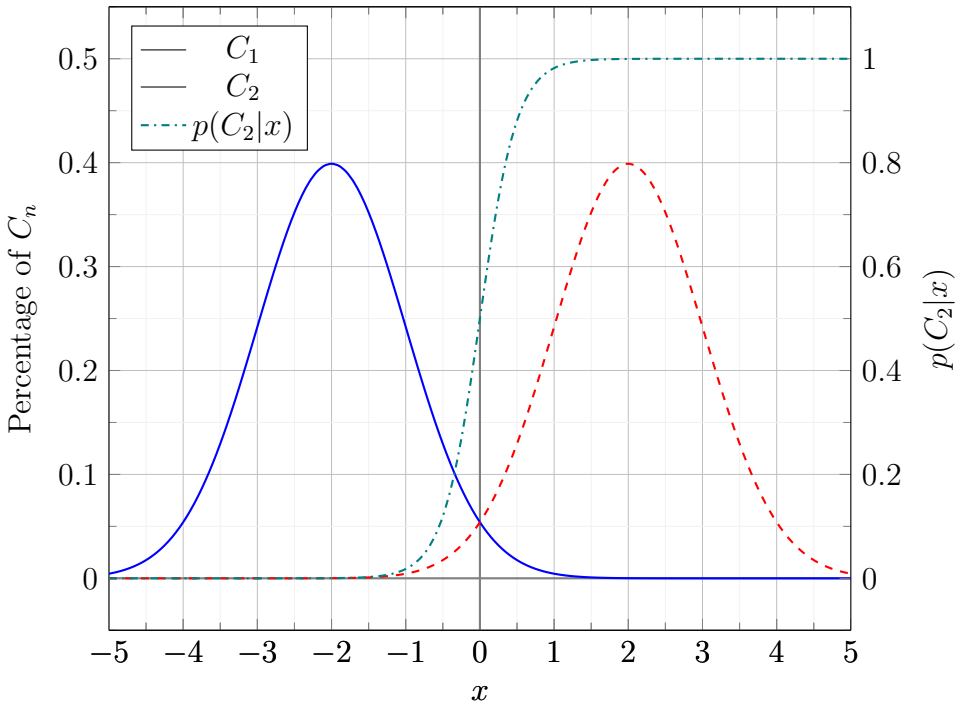


Figure 3.7: Graph showing how two classes with normal distributions (solid blue and dashed red) result in a logistic probability, $p(C_2|x)$ (dot-dashed green). As a result, the logistic sigmoid of Equation 3.6 makes a good probabilistic approximation of the classification problem space.

training uses the differential (or shape) of the activation functions to determine the iterative step size. These two underlying principles make the logistic sigmoid function a sensible choice when attempting to reflect the learning and classification abilities of natural neural systems.

The derivative of the activation function is a critical component when using gradient descent based training algorithms. The logistic sigmoid function has a continuous, non-zero derivative, making it a suitable choice for such training methods as it avoids trapping the system in local minima. Alongside these properties, the logistic sigmoid function also has a simple differential form that may be expressed in terms of the function itself:

$$f'(x) = f(x)\,[1 - f(x)] \tag{3.10}$$

Where $f(x)$ is the logistic sigmoid function and $f'(x)$ is its derivative. This makes calculating the differential a trivial task, providing significant acceleration during the training of large-scale networks.

Despite the advantages of the logistic sigmoid function, it is not without limitations. Most prominent of these is the use of both exponential and divide operations, which make the logistic sigmoid function complicated and resource heavy when implementing it in hardware. Traditional methods for calculating these operations rely on repetitive or iterative solutions that cause non-deterministic timing in hardware systems. Such solutions are also of significant scale when compared against simpler operations, such as add or negate. While this scale and timing difference may seem insignificant when performing a single operation, neural networks require the calculation of many thousands to millions of activation functions in each time-step, resulting in a significant system slowdown if the operations in use are even slightly slower. Equally, the small increase in scale can have considerable compound effects on the size of the whole system if the activation function calculation is parallelised, as is often the case in ANNs.

The asymptotic nature of the function can also result in weight stagnation. The small gradients for input values located at the extremities of the function effectively traps the weights, leading to what is known as the vanishing gradient problem.

3.3.3 Hyperbolic Tangent

The hyperbolic tangent (tanh) function is another logistic function that sees common use in neural network implementations. Unlike the logistic sigmoid function, the tanh function has an output range $tanh(x) \in \{-1, 1\}$. This means that tanh crosses the origin, providing a good approximation for the identity function for small input values. Tanh also has odd symmetry, which is preferred in activation functions as it makes it more likely that the outputs (which in turn forms the inputs of the next neurons) are,

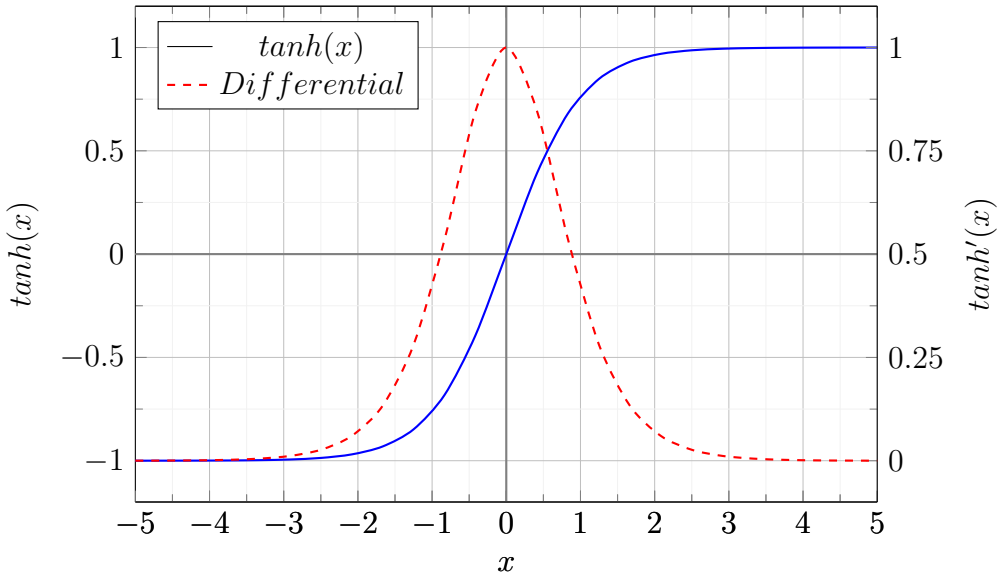


Figure 3.8: The Hyperbolic Tan function (solid blue) and its derivative (dashed red), showing its characteristic logistic S-shaped curve. With an output range $tanh(x) \in \{-1, 1\}$, this function offers odd symmetry, resulting in outputs which are, on average, closer to zero when compared against the similar logistic sigmoid model.

on average, closer to zero [43].

Tanh may be expressed mathematically as follows:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.11}$$

As with the logistic sigmoid function, tanh has a simple differential form that may be expressed in reference to the functions own output. Both the function and its derivative are shown in Figure 3.8.

$$tanh'(x) = 1 - tanh^2(x) \tag{3.12}$$

A comparison of activation functions in classification tasks was performed by Karlik and Vehbi, resulting in the claim that the tanh function performs better than the logistic sigmoid function [44]. In this comparison, a fixed size Multi-Layer Perceptron (MLP) was trained with 500 iterations for each activation function under inspection. This comparison was seemingly performed on one dataset, making it difficult to draw any meaningful or generalised conclusions from the results. The work, however, demonstrates the value of choice when it comes to activation function selection, with different activation functions performing a given task with differing accuracy. Despite the claims of superior performance and the advantages of odd symmetry, tanh is not used as frequently as the sigmoid function due to its greater mathematical complexity.

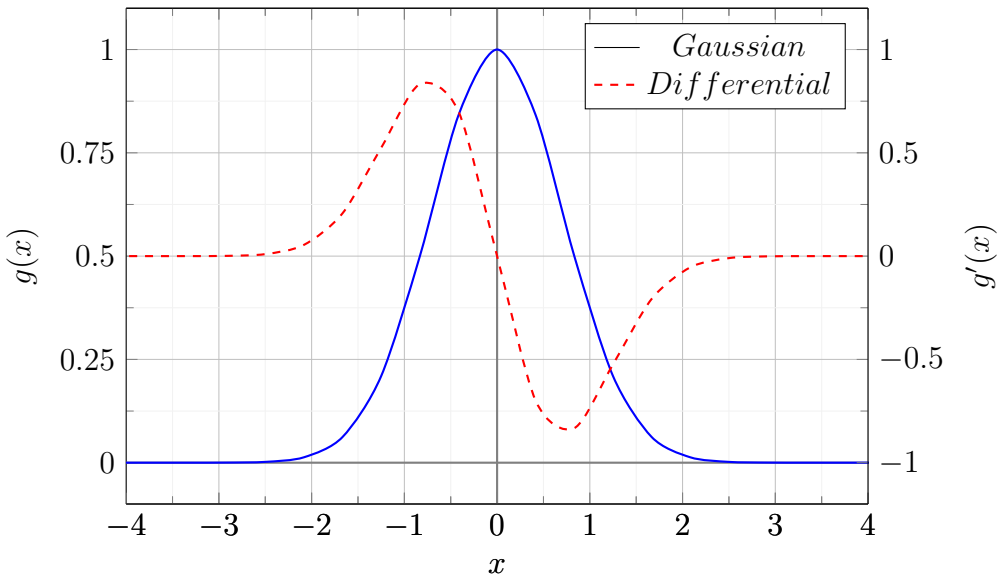


Figure 3.9: The Gaussian function (solid blue) and its derivative (dashed red). While this function is less common, it is easily implemented in hardware and therefore worthy of consideration for mobile or large network arrangements.

Unlike the logistic sigmoid function, tanh requires the calculation of two exponential operations and a divide making it more computationally expensive to implement in hardware.

3.3.4 Gaussian

Alongside the logistic activation functions, the Gaussian function, shown with its derivative in Figure 3.9, has also been used as an activation function in some neural networks. This function has the advantage that it does not require a division, which is typically one of the most expensive operations in most activation functions. Mathematically, the Gaussian function may be defined as follows:

$$g(x) = e^{-x^2} \tag{3.13}$$

As with both the logistic sigmoid function and the tanh function, the Gaussian function has a simple differential form that may be expressed in terms of the functions itself.

$$g'(x) = -2x \cdot g(x) \tag{3.14}$$

Both the Gaussian function and its differential are simpler to implement in hardware than their logistic counterparts, however the Gaussian function has seen only limited

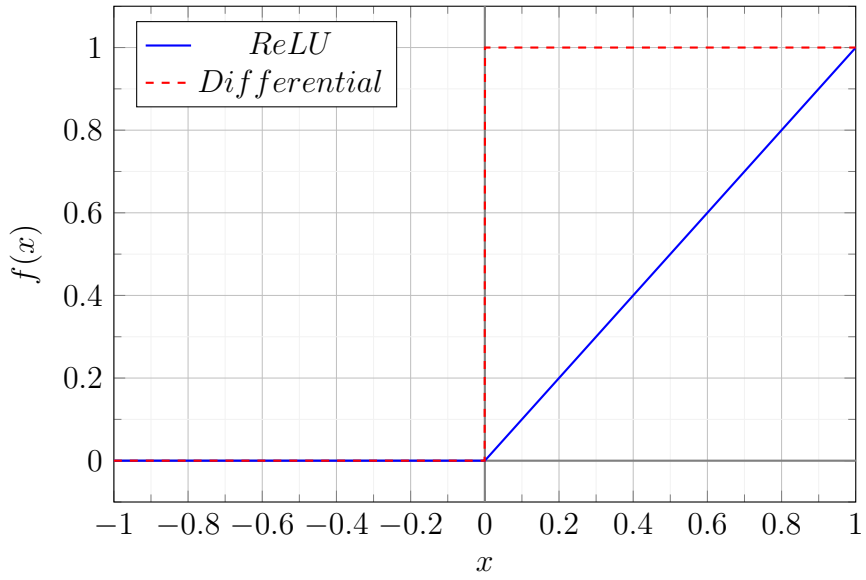


Figure 3.10: The Rectified Linear Unit (or ReLU) function (solid blue) and its step derivative (dashed red). This function is one of the simplest activation functions to implement, making it a popular choice in recent neural network research.

use within the field. This is, in part, due to the fact that the Gaussian function does not offer monotonic behaviour, meaning that two different weights can yield the same performance. This makes training more complicated and can result in slower convergence.

3.3.5 Rectified Linear Unit Function

The rectifier function, also known as the Rectified Linear Unit (ReLU) function, has become the most popular activation function in deep learning and large scale neural network implementations [45, 46]. This function, defined mathematically in Equation 3.15 and shown in Figure 3.10, represents the simplest activation function to implement in hardware, requiring a single sign check and output selection.

$$f(x) = \max(0, x) \tag{3.15}$$

Unlike most of the other activation functions identified in this chapter, the ReLU function does not have a continuous differential, as shown below in Equation 3.16. This can cause issues when performing gradient descent training algorithms. Additionally the zero gradient in the negative region of operation means that neurons using this activation function can easily become stuck in an inactive state, leading to what is known as the dying ReLU problem, where too many neurons become inactive for the

network to perform the desired computation.

$$f'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{3.16}$$

Interestingly, the inactive region of the ReLU function also leads to one of its most significant advantages over other activation functions. During training, any neurons that operate within this negative region produce zero output meaning that they have no influence on the networks operation. As a result, ReLU networks often display greater sparsity than other continuous activation function networks, and the inactive neurons may be safely removed post-training without any impact on the networks performance.

The ReLU function also has the advantage that positive input values will never result in the vanishing gradient problem, however this infinite positive range may easily become an issue as training can cause the weights to run towards infinity if not accounted for within the training algorithm.

Pennington *et. al.* performed an analytical comparison between ReLU and Sigmoidal networks, finding that ReLU networks cannot exhibit dynamical isometry [47]. As a result they show that sigmoid networks can consistently outperform ReLU networks when properly-initialized. In spite of these limitations, the ReLU function has become one of the most popular and actively researched activation functions in the last two years due to its high speed and efficiency when compared against other popular activation functions.

Softplus

A common alternative to the ReLU function is the softplus activation function. This function is sometimes used in place of ReLU to address the issues seen in gradient descent training algorithms [48]. This function provides a smooth approximation of ReLU, resulting in an infinite order of continuity (C^∞). Shown in Figure 3.11, the differential of this ReLU approximation is actually the logistic sigmoid function. The softplus function is mathematically described as follows:

$$f(x) = log(1 + e^x) \tag{3.17}$$

While this approximation provides continuity for gradient descent training, it still has a gradient which rapidly approaches zero for negative input values. This means that the softplus function is still susceptible to the vanishing gradient problem.

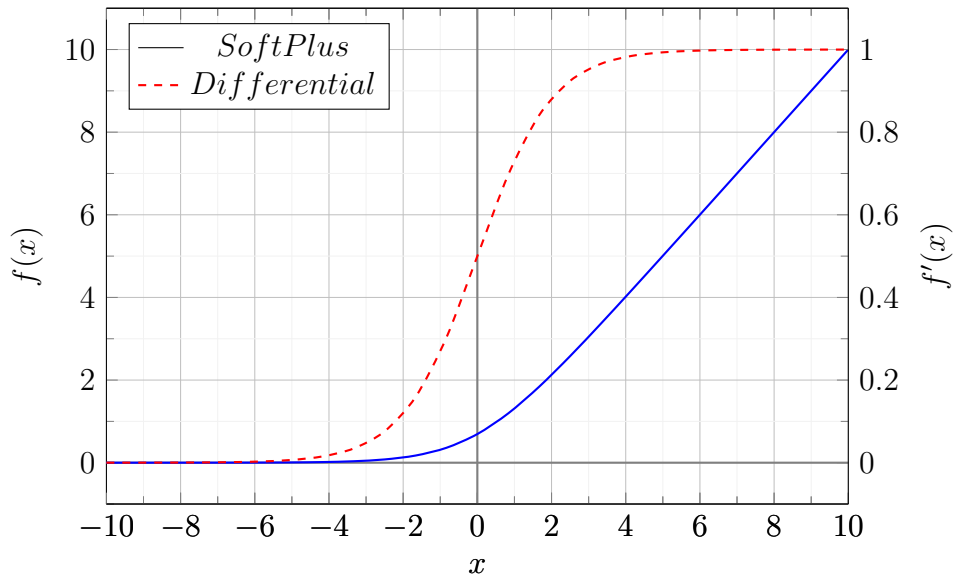


Figure 3.11: The softplus function (Solid blue) and its logistic sigmoid differential (dashed red). This function closely approximates the ReLU function without loosing differential continuity. This improvement comes at the cost of simplicity, requiring considerably more hardware to implement than it's ReLU counterpart.

Leaky ReLU

Introduced by Maas *et. al.* [49] the Leaky ReLU function was designed to address the issues caused by the vanishing gradient problem. Unlike the traditional ReLU, the leaky ReLU adds a small, non-zero gradient to the negative region of the activation function, helping to ensure that it cannot become saturated.

$$f(x) = max(\alpha x, x) \tag{3.18}$$

Where α is a suitably small constant (e.g. 0.1). Such systems were found to perform almost identically to standard ReLU systems when used in Deep Neural Networks (DNNs) [49].

3.4 Neural Network Structures

The ability of a neural network to represent a particular function is related to both the activation function and the network structure. Certain structures are well suited to particular tasks, such as MLP networks, which are typically used for classification tasks; Convolutional Neural Networks (CNNs), which are commonly applied to image recognition tasks; and Recurrent Neural Networks (RNNs), which are commonly applied to temporal information. It is important to select a suitable network structure when

designing a neural network for a given task or problem set. In this section three different structures are highlighted and discussed. These structures were selected to demonstrate specific properties and considerations that influence neural network systems design.

3.4.1 Feed-Forward and Multi-Layer Perceptron Networks

Feed-Forward Neural Networks (FFNNs) are the earliest and simplest form of ANN. Such structures, like the one shown in Figure 3.12, use neurons formed into input, output and hidden layers. The signals travel in a left to right direction from the input layer to the output layer through the hidden layers with no feedback paths.

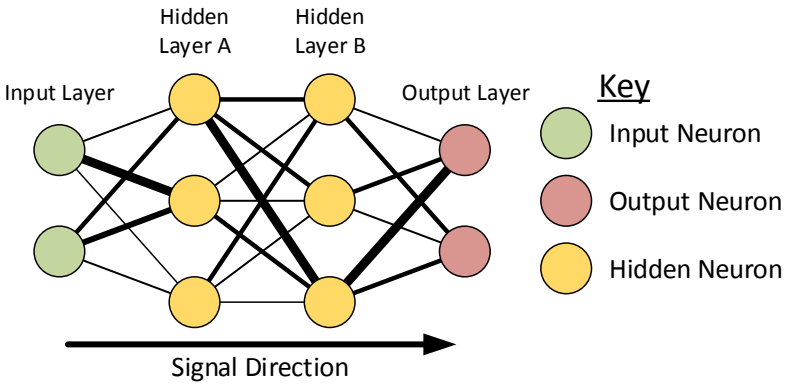


Figure 3.12: Example structure of a four layer feed forward network, showing input neurons, hidden neurons and output neurons. The differing line widths represent different weighting within the networks connections.

MLP networks are a sub class of FFNNs that use the Perceptron model for each neuron. These systems are easy to implement and train making them a practical choice for simple regression, classification and clustering problems. In 1969, Minsky and Papert proved that a single-layer Perceptron network is incapable of computing certain functions (such as XOR) [50]. This rigorous study strongly deterred further neural network research for a number of years and the pessimistic appraisal of neural networks presented in the study has since been heavily criticized [35].

Linear Separability

On reflection, Minsky and Papert's results identified an important property when modelling functions using neural networks. Termed linear separability, this property provides insight into the implementation complexity for a given function. A problem space of n-dimensional vectors is linearly separable if it can be separated with a single

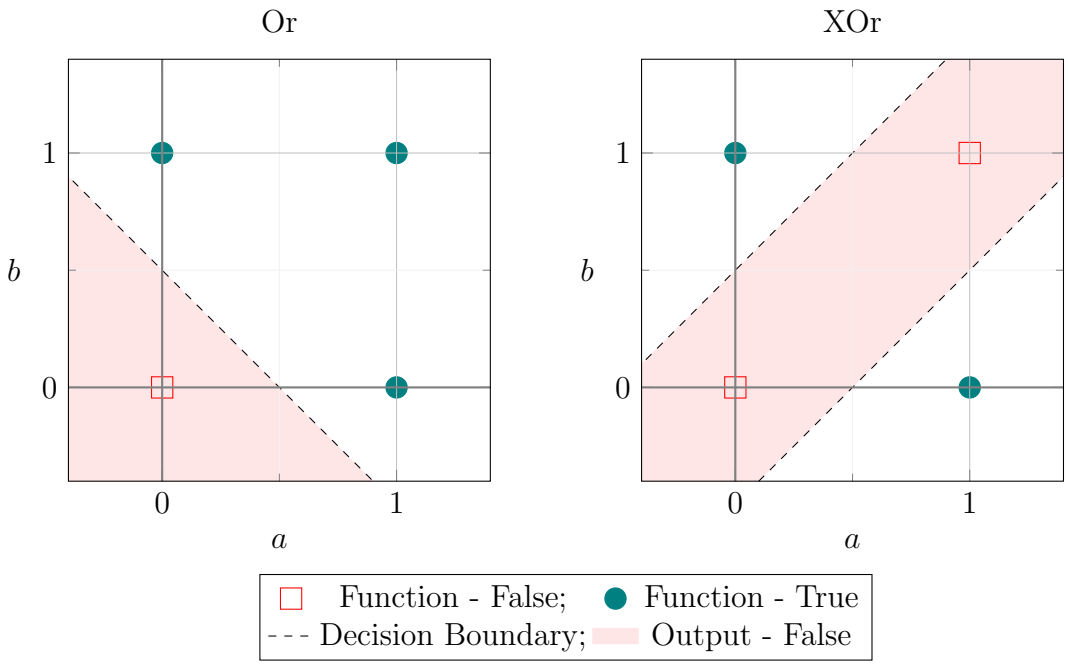


Figure 3.13: The classic linear separability example, showing how the OR function problem space may be divided into it's two output classes using a single line. The XOR function, however, requires two lines to define the problem space, making this function linearly inseparable.

linear decision surface. Figure 3.13 shows two different two-dimensional problems, 'XOR' and 'OR'. The 'OR' problem space may be easily classified using a single decision surface, as shown, and is therefore linearly separable. The 'XOR' problem space, however, is not linearly separable as it requires two decision surfaces to provide the desired output classification.

A single layer Perceptron network, i.e. a single Perceptron, can represent any linearly separable problem so long as there are sufficient inputs for each of the informative dimensions. Multiple linear decision surfaces may be combined using extra layers of Perceptrons to generate more complicated decision surfaces, overcoming this representation limitation.

Network Depth

The number of layers in a network is known as the network depth. Each additional layer in a network allows the previous layers results to be combined in a non-linear fashion producing representations that are otherwise unavailable. A single layer network provides a set of linear decision surfaces. A second layer allows these decision surfaces to be combined with varying weighting, forming convex and open decision shapes. An example problem space is shown in Figure 3.14, with a single decision surface shown for

one layer and a combined decision shape made from two decision surfaces that could be formed using a two layer network. An additional third layer may be used to combine multiple decision shapes, forming complex decision boundaries by overlapping these shapes as shown in Figure 3.14.

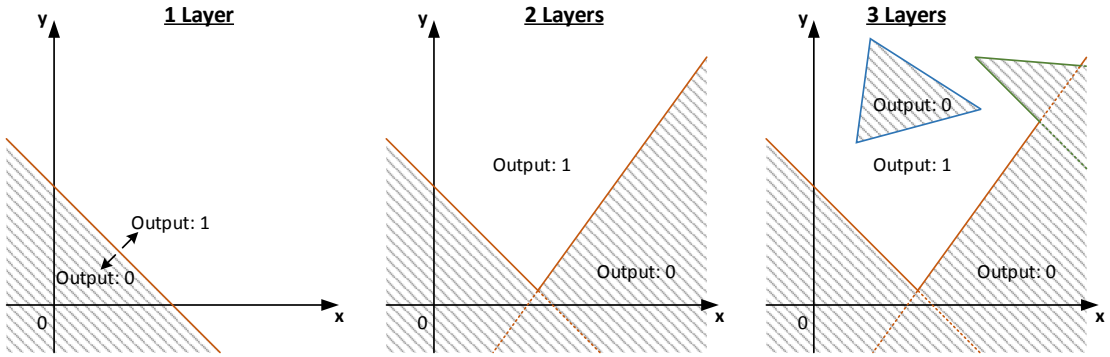


Figure 3.14: Arbitrary problem space division examples achievable by 1-layer, 2-layer and 3-layer FFNNs. A single layer provides a single linear separation. Two layers allows these linear seperations to be combined forming greater decision surface complexity. With three layers these shapes may be combined in an abstract manner, allowing any decision surface shape to be assembled.

The process of combining decision surfaces leads to the result that the linearly inseparable 'XOR' problem can actually be represented by a simple two layer network with two Perceptrons in the first layer and one Perceptron in the second layer, resulting in the desired problem space division shown in Figure 3.13. This demonstrates that linearly inseparable problems may be overcome through the addition of new layers.

The Universality Theorem

The universality theorem states that a three layer neural network can be used to approximate any continuous function with any desired precision and range, so long as sufficient neurons are used in the hidden layers. George Cybenko published the first proof for this concept in 1989 [51] using sigmoid activation to approximate continuous functions. This proof was later generalised by Kurt Hornik in 1991 for multi-layer FFNNs [52]. Figure 3.15 demonstrates the underlying principle, showing an arbitrary function and two approximations generated by adding multiple offset step functions at regular intervals. In this way the approximation may be improved by increasing the frequency and number of step functions used. This principle is very similar to that of the Fourier series, in which a set of sines and cosines may be added together using the superposition principle to generate an arbitrary periodic function. In this example, each step function is equivalent to an output in the hidden layer, so increasing the step function frequency requires the addition of more neurons in the hidden layer.

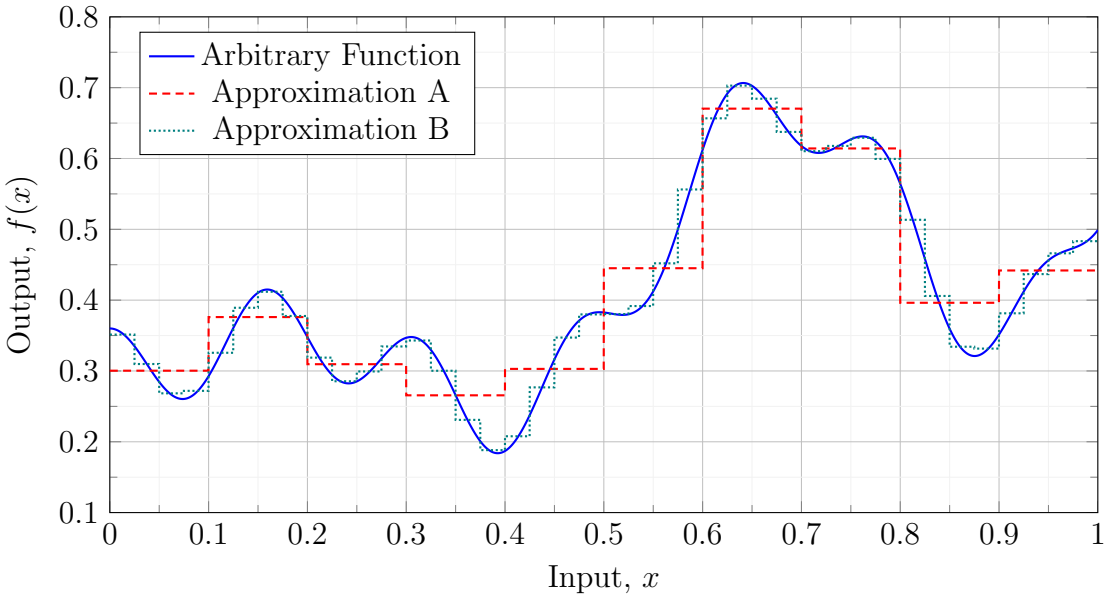


Figure 3.15: An arbitrary function (solid blue) shown alongside two different resolution approximations constructed from time-shifted step functions. Approximation A shows how the general form may be achieved using relatively few step functions, however increasing the frequency (and in turn number of applied step functions) results in greater approximation accuracy as seen in Approximation B.

The hidden layer outputs are then summed by the output layer, yielding the final approximation.

Deep Neural Networks

Despite the universality theorem proving that three layers are sufficient for universal non-linear function approximation, networks with far greater layer counts have still become a key component of research efforts in the last 5 years. Such networks are termed Deep Neural Networks (DNNs) in reference to the depth of layers found within their internal structure. These DNNs can model complex non-linear relationships within the provided input data, allowing complicated classification and regression tasks to be performed. The extra layers enable the system to identify features in previous layers, supporting associative abstraction of the data. Thus the network can model complex data with fewer computational units than would be required by an equivalent shallow network [53].

The addition of extra layers makes training more challenging, often resulting in computationally expensive training procedures. Despite this issue, significant advancements within the field of artificial intelligence have been provided by DNNs, and with computational power continuing to improve it seems likely that such networks will continue to find application in a range of specialisms [54].

3.4.2 Recurrent Neural Networks

Unlike FFNNs, RNNs support bi-directional data flow within their internal structure. This allows RNNs to learn patterns in sequential information, using historic entries to influence the current network state. These networks may be considered to contain a form of internal memory, represented by feedback loops within the networks themselves.

The temporal nature of these networks makes them particularly suitable for processing and classifying sequential data such as: in language modelling, where the aim is to measure the likelihood of a sentence structure or occurrence [55, 56, 57]; machine translation problems, providing automated language translations [58, 59]; speech recognition [60, 61, 62]; and musical composition [63].

Training RNNs presents a challenge, due to the hidden internal memory and signal delays. The influence of internal memory on the systems state isolates the networks instantaneous error from the current input values making it difficult to produce a quantitative performance metric. Equally, the temporal nature of these systems means that a correct output could become incorrect with time, making it challenging to test a networks performance within a reasonable time frame. Techniques such as 'back-propagation through time' [64] have been developed to enable the training of such systems, however these operations can take considerable time and computational resource. Many of these methods also suffer from the vanishing gradient problem previously mentioned, where small gradients in the error space trap the system parameters in a non-optimal solution.

One of the most common forms of RNN is the Long Short-Term Memory (LSTM) network. These networks are better at capturing long-term dependencies than other RNNs and avoid the vanishing gradient problem. LSTMs were first proposed by Hochreiter and Schmidhuber in 1997 [65] and have achieved notable results in a range of tasks, such as handwriting recognition [66].

3.4.3 Hopfield Networks and Boltzmann Machines

So far, the networks considered have all contained sparse connectivity in the form of layers, with distinct and separate input and output neurons. Hopfield networks, however, do not follow this pattern. These networks are a form of RNN popularised by John Hopfield in 1982 [67] but first described by Little in 1974 [68]. Unlike other structures, Hopfield networks make use of fully connected neurons with no hidden layers. They implement I/O neurons which act as both input and output for the network at the same time.

Hopfield networks operate as content-addressable or associative memory. Each neuron is fixed with an input value before the network is set free and allowed to oscillate. After a suitable time period the network converges upon one of its learnt patterns, producing

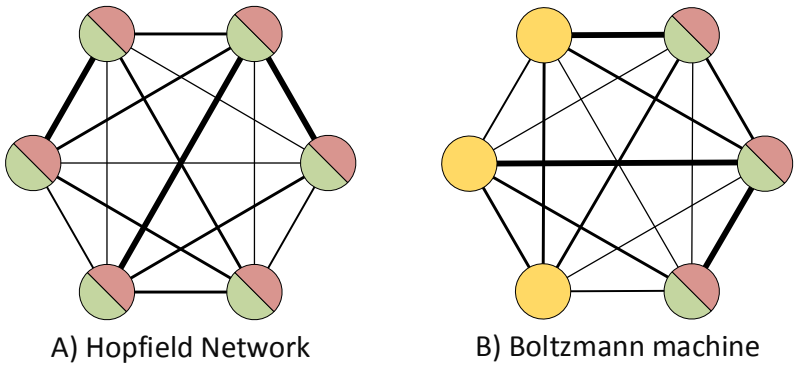


Figure 3.16: Example structures for both Hopfield networks (a) and Boltzman machines (b). Hidden neurons are shown in solid yellow circles, while I/O neurons are represented using bisected red-green circles. The varying edge thickness represents the varying weights used in such networks.

steady outputs on the I/O neurons. In this way the network is capable of generating a full pattern when only partial data is available, much like how human memory can trigger detailed recollection following a partially associated stimulus.

Hopfield originally used binary threshold neurons [67]. In later work, Hopfield developed an associative memory with continuous outputs in the range $y \in \{-1, 1\}$ [69].

One issue with Hopfield networks is that they tend to stabilize to local minima rather than the global minimum. This is largely addressed by a class of networks called Boltzmann machines, which are trained using statistical rather than deterministic methods [70]. Due to the close analogy between this statistical training approach and metal annealing, these methods are commonly called 'simulated annealing'. Unlike Hopfield networks, Boltzmann machines make use of hidden neurons within their structure. A global temperature is often implemented to avoid continuous oscillation of these networks. Lowering this global temperature acts to lower the neuron energies, in turn encouraging stabilisation of the network.

3.4.4 Convolutional Neural Networks

While FFNNs and RNNs have historically dominated neural network research, CNNs have become some of the most popular and widely applied neural networks within the early part of the 21st century. Also known as shift invariant and space invariant artificial neural networks, CNNs were heavily inspired by the visual cortex in animals. In 1968, Hubel and Wiesel noted that the visual cortex of monkeys was constructed in a structured and identifiable manner, with only two key visual cell types in the brain [71]. Building upon this concept, LeCun developed LeNet-5, the first artificial CNN [72]. Since their introduction CNNs have found considerable use in tasks such as image recognition, frequently outperforming other neural network topologies. Rawat and

Wang provide a comprehensive review of CNN implementations in image classification tasks. Despite its biological inspiration, after reviewing around 300 publications and identifying a number of state of the art CNN results Rawat and Wang conclude that the most significant challenge facing the research community is still to be found in closing the theoretical gap between biological neural networks and CNNs [5].

A CNN typically consists of an input layer, output layer and many hidden layers. Each of these hidden layers will commonly fall into one of three categories: convolutional layers, pooling layers and fully connected layers. Each type of layer performs a specific role and the arrangement of these layers will often impact the final performance of the network.

Convolutional layers perform the shift invariant feature recognition. The result of these layers is equivalent to convolving a filter or mask over the input and, as a result, the output will always match the dimensions of the input. For image processing CNNs this means that each convolutional layer produces a filtered image of its own. This convolutional filtering is performed though use of many identical neurons that all have the same weights as one another. In this way the convolutional layers are said to use 'shared weights', greatly reducing the number of free parameters that must be optimised during training. These shared weight neurons are located spatially across the full data space and only operate on a small local neighbourhood termed the 'receptive field'. As a result these neurons operate in a spatially local domain, requiring no long distance or global connections. Figure 3.17 shows a small selection of convolutional neurons in a single convolutional layer. The weights of each of these neurons are identical (represented using matching colours) regardless of where on the image the neuron operates.

Pooling layers compress the data, reducing the number of data points through a simple operation. Max pooling selects the maximum value from a region on the previous layer, while average pooling uses the average value from the previous layer. Both these methods are illustrated in Figure 3.18.

Fully connected layers are often located at the end of the CNNs structure and operate in the same manner as that of a perception neural network. These layers perform the classification task, operating in a similar way to other classification networks.

In practice these layers are used in groups, such that there are many convolutional filters applied to the previous layers outputs at the same time/stage. This means that a 2D image input becomes a 3-dimensional data-structure within the CNN. In the case of classification tasks, the fully connected layers receive this 3D structure, processing it into a 1D list of class likelihoods. The convolutional layers may be visualised to show the features that are identified by each filter, as shown in Figure 3.19. From this figure it is seen that the first filters act as edge detectors, with the feature recognition becoming ever more advanced as the data is passed to additional layers.

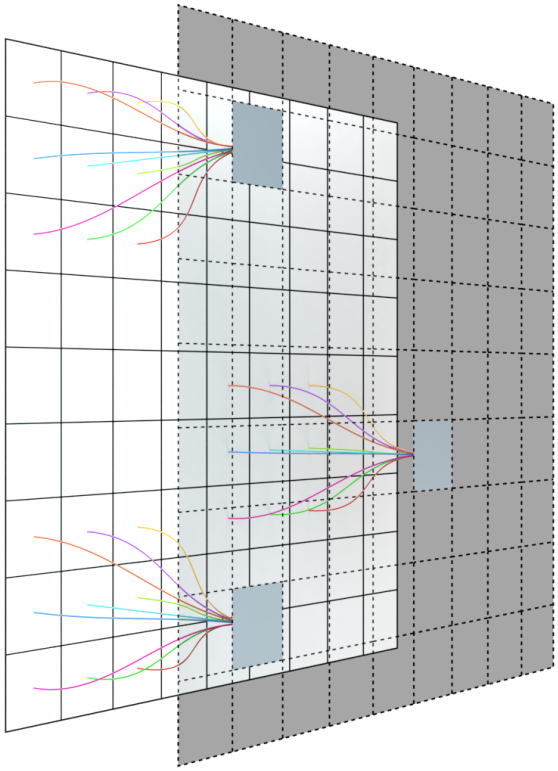


Figure 3.17: Example convolutional layer, showing three neurons neighbourhoods on the previous layer. Shared colours of the connection lines represent the shared weights which are consistent between each of the neurons in the convolutional layer.

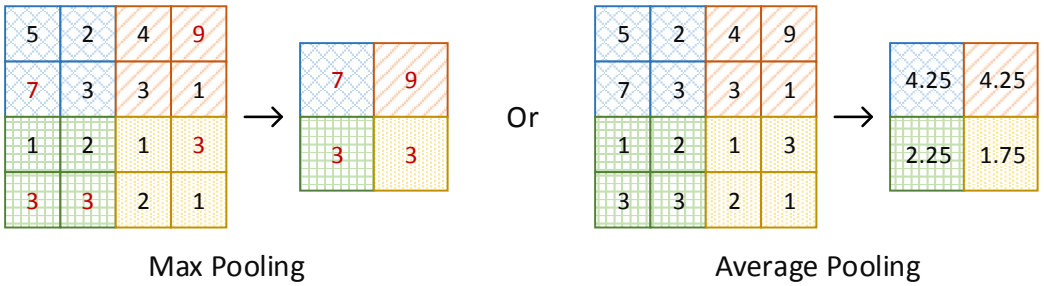


Figure 3.18: Examples of max pooling and average pooling operations commonly used in the pooling layers of CNNs. These layers help reduce the resolution of the input data, allowing the system to identify the large structures within the data.

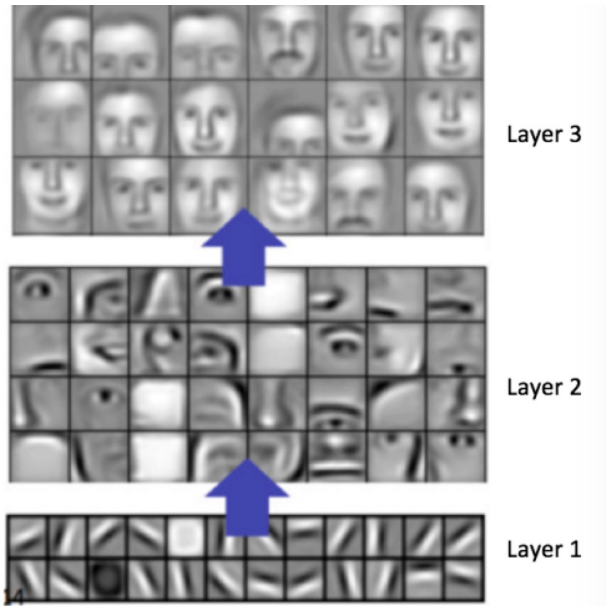


Figure 3.19: Example CNN layers taken from *Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations* by H. Lee [73], showing how the early convolutional layers form basic edge recognition, with the identified features growing in complexity as the network is traversed deeper.

3.5 Network Training

One of the key factors of neural networks popularity is found in their ability to learn and adapt to new problems or tasks. This learning process is apparent within many biological systems and offers considerable computational value if successfully mirrored in artificial systems. Significant progress has been made within this area, however there are still unanswered and fundamental questions when it comes to the nature of learning.

Typically neural network learning and training may be classed in one of two distinct categories: offline and online learning. When using online learning, the network will continue to refine and improve its classifications using the live data provided. Such systems often require an oracle or feedback path to provide a measure of success and error. In offline learning, systems are trained using a pre-defined set of data before locking the internal weights and properties for use in a live environment. This method often requires the data to be pre-processed to ensure that the correct result is available for comparison and validation of performance.

Network training may be further classified into supervised and unsupervised learning. Supervised learning is performed when the result or desired outcome is already well known. In this case the network is trained against the desired result to ensure that it learns the underlying rules of the problem. Typically these tasks will either be classification or regression problems. In classification tasks the data must be assigned

to categories (e.g. cat or dog), while regression tasks produce a real value (e.g. weight).

In unsupervised learning the network must attempt to find some underlying pattern or rule set that was previously unknown. Such systems will often be designed to perform clustering or association tasks. A clustering problem requires the network to discover an inherent grouping for the input data (e.g. online profiling through user behaviour), while an association problem requires the network to discover the fundamental rules that describe the bulk data provided (e.g. popular similar items in an online store).

Both supervised and unsupervised learning occur in biology, however supervised learning has proven significantly easier to implement in artificial systems due to the inherent measure of accuracy or error it provides.

3.5.1 Training, Test and Validation Datasets

Datasets play a key role in ANN training and validation processes, and the way these datasets are formed and used is critical to the success or failure of a network. It is common practice to separate the data into three unique non-overlapping datasets, these are the training data, test data and validation data. Each dataset must contain a random selection of data points such that the sets suitably represent the problem space.

The training and test datasets are used during the training cycles. First, the training data is provided to the network and a training algorithm is iteratively used to tweak the networks parameters and reduce error. The test data is used between iterations to generate a measure of network performance during training. The training algorithm is switched off while running the test data through the network. The separation of training and test data helps ensure that the network does not simply learn and mirror the raw data used during training, in a condition termed *over-fitting*. As shown in Figure 3.20, over-fitting results in poor test data performance when compared against the training data performance. It is important to stop training before over-fitting to ensure that the network performs optimally on unseen data.

Validation data may be used post-training to provide a measure of network performance for comparison with other solutions. This validation data-set contains previously unseen data, ensuring that the training process has not simply optimised the network for the training and test datasets provided. All three datasets are critical components in ensuring that a network will continue to achieve minimal error when presented with live and unseen data.

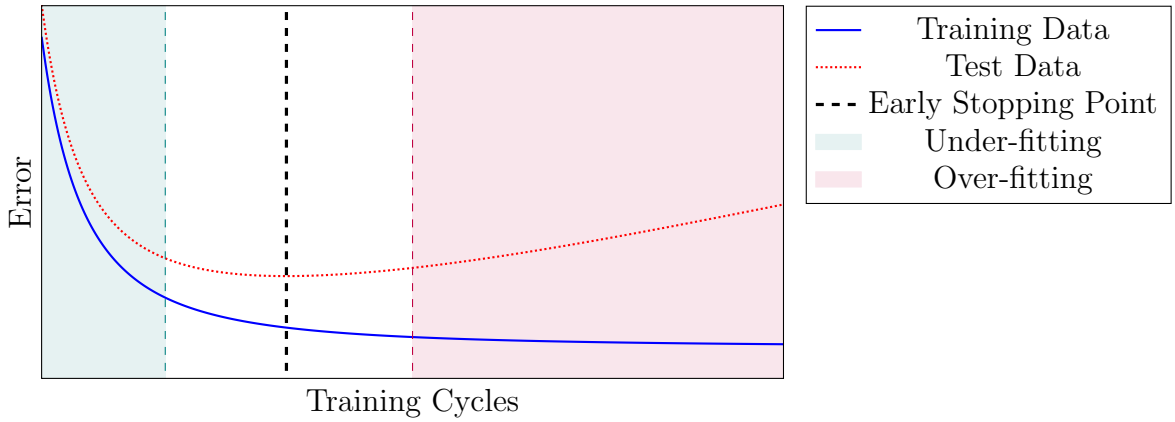


Figure 3.20: Example error seen during ANN training, showing how the networks training data performance continues to improve with additional training cycles. Despite this apparent improvement, the test data results reveal that the networks performance on unseen data actually becomes worse after the early stopping point due to over-fitting.

3.5.2 Hebbian learning

Hebbian theory is an important neuroscience theory developed by Donald Hebb in 1949 [37]. As described in Section 3.2.1, Hebb's rule states that any neurons that repeatedly play a role in firing another neuron will result in growth that emphasises this trait. Hebbian theory is also found in the idea that any two neurons (or indeed groups of neurons) that are frequently active simultaneously will, with time, become associated to one another such that one may encourage activity in the other. Mathematically this is often generalized into the form:

$$\Delta w_i = \eta x_i y \tag{3.19}$$

where y is the postsynaptic response and w_i is the ith weight, which modifies the ith input, x_i. The learning rate, η, controls the impact of learning on the weights and is often small to avoid oscillation and ensure that any significant change is as a result of the bulk behaviour.

This learning rule is simple to implement when considering single neurons or synapses, however it does not monitor the performance of the network and therefore is not suitable as a training algorithm on it's own.

3.5.3 Back-Propagation Algorithm

The training of multi-layer artificial networks presented a significant problem in the advancement of neural networks. Combined with Minsky and Papert's critical review

of neural networks [50], the lack of any theoretically sound algorithm for training such systems led to a severe decline in neural network research. The invention of the back-propagation algorithm played a considerable role in the resurgence of neural network research, finally offering a methodology for training multi-layer networks.

Presented by Rumelhart, Hinton and Williams in 1986 [74], the back-propagation algorithm has something of a contested past. Shortly after publication, Parker was shown to have anticipated the work in 1982 [75]. After this it was discovered that Werbos had actually detailed the method as early as 1974 [76]. Despite its contested origins, the back-propagation algorithm has been key to expanding the range of problems that neural networks may be successfully applied to.

Defining Network Error

The aim of back propagation is to reduce a networks overall error by manipulating individual weights within the network. As such it is important to first define how this overall error is calculated. For a given input vector the networks error, E_{total}, is commonly found using the Mean Squared Error (MSE) of Equation 3.20, where T and Y are the target output and actual output values respectively for each of the m neurons in the output layer.

$$E_{total} = \sum_{i=1}^{m} \frac{1}{2}(T_i - Y_i)^2 \tag{3.20}$$

Considering a single neuron

In order to comprehend the back propagation algorithm it is necessary to first consider the forwards function of a single neuron, shown in Figure 3.21.

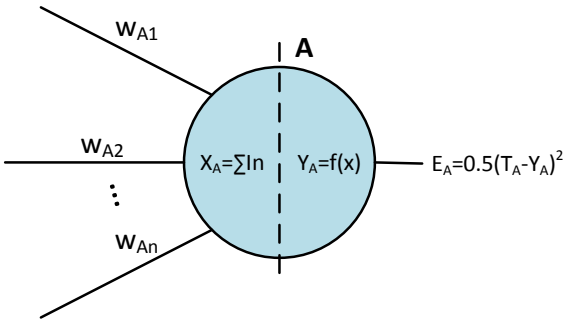


Figure 3.21: Internal operation of a single integrate and fire neuron. The internal process is shown as two distinct steps, allowing derivation of back propagation algorithm independent to activation function selection.

The neuron may be considered to have two separate internal functions. First, the weighted inputs are summed and the resultant value, x, is then passed through the activation function, $f(x)$. By separating the neurons internal functions in this way it becomes possible to swap out the activation function used within the back-propagation algorithm. If the neuron is located on the output layer of the network the error may be found using equation 3.20.

The Output Layer

As the name suggests, the back propagation algorithm functions by propagating the error backwards through the network, correcting the internal weights one layer at a time as it progresses. As such it is necessary to start at the output layer of the network.

Considering a single output layer neuron such as the one shown in Figure 3.21, the change in error, E_{total}, caused by changing a given weight, $W_{modified}$, may be written mathematically as $\partial E_{total}/\partial W_{modified}$. Using the chain rule, this differential may divided into the same sub-parts used in Figure 3.21, as shown in Equation 3.21.

$$\frac{\partial E_{total}}{\partial W_{modified}} = \frac{\partial E_{total}}{\partial y_A} \cdot \frac{\partial y_A}{\partial x_A} \cdot \frac{\partial x_A}{\partial W_{modified}} \tag{3.21}$$

Each of the sub-parts in Equation 3.21 may be found separately, before combining them again to yield the desired derivative.

The Output Layer - Part 1

Using the definition of network error in Equation 3.20, the derivative of the error w.r.t. a given neurons output may be written as:

$$\frac{\partial E_{total}}{\partial y_A} = \frac{\partial \sum_{i=1}^{m} \frac{1}{2}(T_i - y_i)^2}{\partial y_A} \tag{3.22}$$

The differential of the sum elements will be zero when $A \neq i$. This may therefore be simplified to Equation 3.24 as follows.

$$\frac{\partial E}{\partial y_A} = \frac{\partial \frac{1}{2}(T_A - y_A)^2}{\partial y_A} \tag{3.23}$$

$$\frac{\partial E}{\partial y_A} = y_A - T_A \tag{3.24}$$

The Output Layer - Part 2

Considering an arbitrary activation function $f(x)$, $\partial y_A / \partial x$ may simply be rephrased as shown in Equation 3.25.

$$\frac{\partial y_A}{\partial x_A} = f'(x_A) \tag{3.25}$$

The Output Layer - Part 3

Finally, the derivative of x w.r.t. the modified weight, $W_{modified}$, may be found using Equation 3.26 where y_{prev_i} is the output of neuron i in the previous layer and p is the total number of neurons in the previous layer.

$$\frac{\partial x_A}{\partial W_{modified}} = \frac{\partial \sum_{i=1}^{p} (W_i \cdot y_{prev\_i})}{\partial W_{modified}} \tag{3.26}$$

As with Equation 3.22, the elements of this sum which do not use $W_{modified}$ will have no partial derivative. This yields the simplified form shown in Equation 3.28, with $y_{modified}$ referencing the output of the neuron on the previous layer which is connected to the weight under modification.

$$\frac{\partial x_A}{\partial W_{modified}} = \frac{\partial (W_{modified} \cdot y_{modified})}{\partial W_{modified}} \tag{3.27}$$

$$\frac{\partial x_A}{\partial W_{modified}} = y_{modified} \tag{3.28}$$

The Output Layer - Combined

Combining each of the sub-parts in Equations 3.24, 3.25 and 3.28, the impact of modifying a given weight on the networks overall error may be found as shown in

Equation 3.29.

$$\frac{\partial E_{total}}{\partial W_{modified}} = (y_A - T_A) \cdot f'(x_A) \cdot y_{modified} \tag{3.29}$$

Using this calculated impact factor, a new weight may now be calculated using Equation 3.30.

$$W_{new} = W_{modified} - (\eta \times \frac{\partial E_{total}}{\partial W_{modified}}) \tag{3.30}$$

where, η is the learning rate parameter and W_{new} is the post-modification weight.

The Hidden Layer

Having calculated the new weights for the output layers, the back propagation algorithm must then advance to the previous layer in the network. This layer does not have a direct one-to-one relationship with the output error of the network and it is therefore again necessary to calculate how changes in this layer will affect the overall error. A simple example of this problem is shown in Figure 3.22, where the output of the target neuron A passes through two separate output neurons, B and C, before contributing to the overall error.

As before, the change in error, E_{total} w.r.t. a given weight, $W_{modified}$ must be calculated. Applying the chain rule and using the elements in Figure 3.22 yields the following equation.

$$\frac{\partial E_{total}}{\partial W_{modified}} = \frac{\partial E_{total}}{\partial y_A} \cdot \frac{\partial y_A}{\partial x_A} \cdot \frac{\partial x_A}{\partial W_{modified}} \tag{3.31}$$

The latter two sub-parts of Equation 3.31 are of identical form to those found for the output layer neurons. Therefore the only part which must be newly defined is that of the change in error w.r.t. the neurons own output.

As shown in Figure 3.22 the output of the neuron may contribute to a number of different error values (such as E_B and E_C). The error w.r.t. output may therefore be written as follows.

$$\frac{\partial E_{total}}{\partial y_A} = \sum_{i=1}^{m} \frac{\partial E_i}{\partial y_A} \tag{3.32}$$

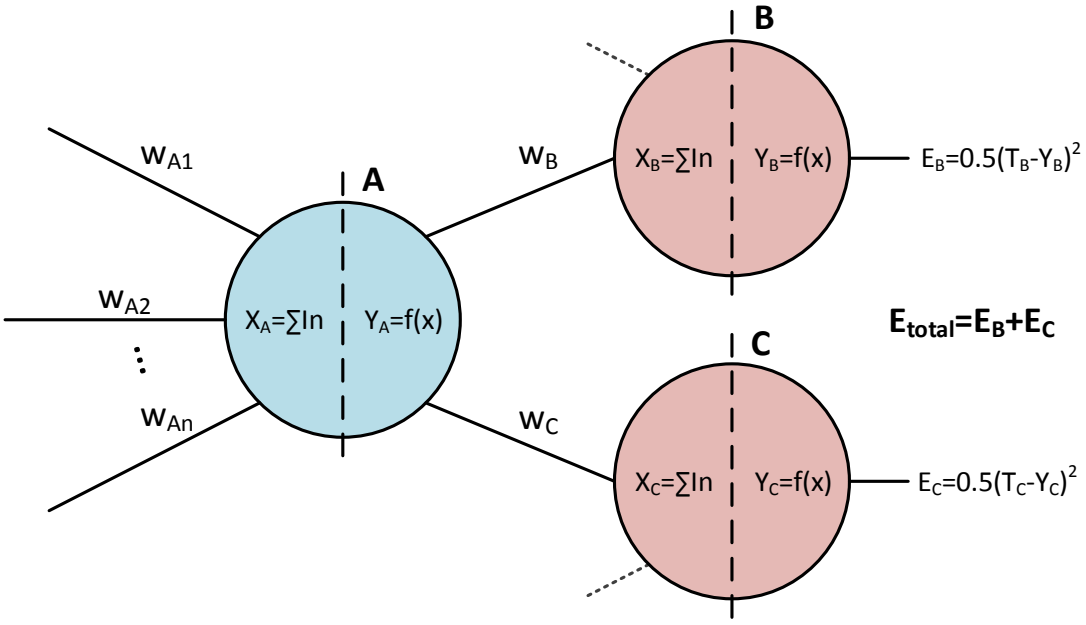


Figure 3.22: Layered neurons used to calculate the influence of early layer weights on the final output error of the system. It may be seen that a weight change in the first layer (A) will propagate through the second layer neurons (B and C), modifying the final result of both E_B and E_C.

Applying the chain rule for a second time this may be re-written using the sub-parts shown in Figure 3.22.

$$\frac{\partial E_{total}}{\partial y_A} = \sum_{i=1}^{m} \left(\frac{\partial E_i}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_i} \cdot \frac{\partial x_i}{\partial y_A} \right) \tag{3.33}$$

Two of these three parts have been previously defined, yielding the expanded format shown below in Equation 3.34.

$$\frac{\partial E_{total}}{\partial y_A} = \sum_{i=1}^{m} \left((y_i - T_i) \cdot f'(x_i) \cdot \frac{\partial x_i}{\partial y_A} \right) \tag{3.34}$$

Considering the final part of Equation 3.34, it may be shown that $\partial x_i / \partial y_A = W_i$, where W_i is the weight between neuron i of the output layer and neuron A in question. This yields a final equation for error w.r.t. output as shown below in Equation 3.35.

$$\frac{\partial E_{total}}{\partial y_A} = \sum_{i=1}^{m} ((y_i - T_i) \cdot f'(x_i) \cdot W_i) \tag{3.35}$$

It should be noted at this stage that Equation 3.35 represents the impact of output changes at the current layer as they are fed through the rest of the network layers. Using Equation 3.29, it is possible to re-define this into a form which is irrespective of the neurons depth within the network, as shown below. In this new form the results from the previously calculated layers are used to calculate the results for the current layer, allowing rapid progression through the network.

$$\frac{\partial E_{total}}{\partial y_A} = \sum_{i=1}^{m} \left(\frac{\partial E_{total}}{\partial W_i} \cdot \frac{W_i}{y_i} \right) \tag{3.36}$$

Applying this expansion to Equation 3.31 yields a suitable equation for finding the impact of modifying a weight buried within a network, as shown below in Equation 3.37. Once again, $y_{modified}$ refers to the value which the weight under modification would scale in normal operation, while r represents the number of neurons in the previously calculated layer.

$$\frac{\partial E_{total}}{\partial W_{modified}} = \sum_{i=1}^{r} \left(\frac{\partial E_{total}}{\partial W_i} \cdot \frac{W_i}{y_A} \right) \cdot f'(x_A) \cdot y_{modified} \tag{3.37}$$

Algorithm Summary

Once each of the weight differentials have been defined the overall back propagation method may be easily applied to any network. For ease of reference the key equations are repeated in this section, with notes on their application where required.

The output layer weight differentials and other layer weight differentials may be found using Equations 3.38 and 3.39 respectively.

$$\textbf{Output Layer: } \frac{\partial E_{total}}{\partial W_{modified}} = (y_A - T_A) \cdot f'(x_A) \cdot y_{modified} \tag{3.38}$$

$$\textbf{Other Layers: } \frac{\partial E_{total}}{\partial W_{modified}} = \sum_{i=1}^{r} \left(\frac{\partial E_{total}}{\partial W_i} \cdot \frac{W_i}{y_A} \right) \cdot f'(x_A) \cdot y_{modified} \tag{3.39}$$

Since Equation 3.39 relies on the previous set of results, it is necessary to work through the network one layer at a time, starting with the output layer. Using the newly calculated weight differentials, a new weight may now be calculated for each layer using Equation 3.30. Care must be taken to ensure that the original weights are used for any

further calculations of weight differentials in a single back propagation training cycle. The change or correction to each weight may therefore be defined as follows:

$$\Delta W_i = -(\eta \times \frac{\partial E_{total}}{\partial W_i}) \tag{3.40}$$

where η is the learning rate.

3.5.4 Advanced Gradient Descent Algorithms

When using gradient descent training algorithms such as back-propagation there is a trade-off to be considered when selecting the learning rate for the system. Steep problem space gradients result in oscillation for systems with large learning rates, however smaller learning rates will cause the system to propagate slowly in shallow regions of the problem space. It is common to select smaller learning rates since an oscillating system may never find a stable solution. For this reason, back-propagation and other simple gradient descent algorithms are commonly found to converge slowly when training. A number of different methods have been developed to address this issue. These approaches often modify the step sizes on each iteration of the algorithm, attempting to select the optimal learning rate using extra information about the problem space itself.

Newton's Method

Newton's method (also known as Newton-Raphson iteration) is a common numerical analysis tool for incrementally approximating the roots of a real-valued function. This method provides quadratic convergence and has seen considerable use in a number of applications, such as calculating the reciprocal of a number in hardware systems. The quadratic convergence of this method means that it can outperform most other methods when convergence is possible. The convergence itself, however, is not guaranteed if the starting value is too distant from the actual solution. For this reason the underlying function and starting values must be carefully considered if Newton's method is to be successfully applied.

In it's simplest form, Newton's method may be stated as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.41}$$

where x_n is the nth approximation of a root and $f(x)$ is the real-valued function under consideration.

This function provides an iterative method for finding the roots or zero crossing values. In the case of the reciprocal function, this may be performed by first defining $f(x)$ such that $f(1/y) = 0$. The Newton method may then be used to find the root of $f(x)$ which is, in turn, the reciprocal of y.

This method may also be used in optimisation problems, such as neural network training. It is possible to optimise a system by solving the differential since a root or zero crossing in the differential of a function represents a maxima or minima in the function itself. Therefore Newton's method may be applied to the differential of the error space, yielding the minima locations. To perform this operation requires calculation of the second-order derivative for the error space, and this method is therefore classed as a second order algorithm.

This approach may be generalised for several dimensions by replacing the derivative of the problem space with the gradient of the problem space, $\nabla f(\mathbf{x})$, and the reciprocal of the second derivative with the inverse of the Hessian matrix, $\mathbf{H}(f(\mathbf{x}))$, where the Hessian matrix may be defined as:

$$\mathbf{H}(f(\mathbf{x}))_{i,j} = \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_j} \tag{3.42}$$

where i and j are the indices for the matrix.

These modifications yield the following version of Newton's method:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \ \left[\mathbf{H}(f(\mathbf{x}_n))\right]^{-1} \nabla f(\mathbf{x}_n) \tag{3.43}$$

where $\gamma \in (0, 1)$ to ensure smaller step sizes.

While this method provides better convergence over general gradient descent algorithms, the calculation of the Hessian matrix can be very computationally intensive. This makes this method unsuitable for many optimisation problems.

Gauss-Newton algorithm

The Gauss-Newton algorithm is a modified version of the Newton method that can only solve non-linear least squares problems. Despite this limitation, the Gauss-Newton algorithm has the considerable advantage that it does not require the second derivatives to be calculated.

Given m functions $\mathbf{r} = (r_1, \ldots, r_m)$ with n variables $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_n)$ where $m \geq n$, the Gauss-Newton algorithm attempts to find the values of $\boldsymbol{\beta}$ that minimises the sum

of the squares, $\mathbf{S}(\boldsymbol{\beta})$, given by:

$$\mathbf{S}(\boldsymbol{\beta}) = \sum_{i=1}^{m} r_i^2(\boldsymbol{\beta}) \tag{3.44}$$

The iterative approximations for $\boldsymbol{\beta}$ may then be found using the following equation:

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \left(\mathbf{J_r}^\top \mathbf{J_r}\right)^{-1} \mathbf{J_r}^\top \mathbf{r}\left(\boldsymbol{\beta}^{(k)}\right) \tag{3.45}$$

where $\boldsymbol{\beta}^{(k)}$ is the kth approximation and $\mathbf{J_r}$ is the Jacobian matrix, such that:

$$\left(\mathbf{J_r}\right)_{i,j} = \frac{\partial r_i\left(\boldsymbol{\beta}^{(i)}\right)}{\partial \beta_j} \tag{3.46}$$

While this method requires less computation than that of the Newton algorithm, there is no guarantee that it will actually converge on a solution. Indeed this algorithm will converge slowly, or even not at all, if the starting point is too far from a minimum. Equally, if the matrix $\mathbf{J_r}^\top \mathbf{J_r}$ is ill-conditioned convergence may fail. This lack of convergence creates problems when attempting to optimise large neural networks with unknown problem spaces.

Levenberg-Marquardt algorithm

The Levenberg-Marquardt algorithm was independently developed by Levenberg [77] and Marquardt [78]. Also designed to solve non-linear least squares problems, this algorithm forms a hybrid of the Gauss-Newton algorithm and standard gradient descent techniques. As a result, the Levenberg-Marquardt algorithm is more robust than the Gauss-Newton algorithm, meaning it often finds a solution even when initialised far away from the final result. By leaning on the strengths of the Guass-Newton algorithm, this method provides an accelerated convergence when compared with traditional gradient descent techniques.

Suitable for small and medium sized neural network problems, the Levenberg-Marquardt algorithm interpolates between the Gauss–Newton algorithm and gradient descent methods. Favouring standard gradient descent for regions that contain complex curvature and utilising the Gauss-Newton method for areas where the problem space is such that it is proper to make a quadratic approximation [79].

The algorithm makes use of an approximation to the Hessian Matrix, defined as:

$$\mathbf{H} \approx \mathbf{J}^\top \mathbf{J} + \mu \mathbf{I} \tag{3.47}$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{J}$ is the Jacobian matrix and μ is a positive value called the combination coefficient.

With these approximations defined, Equation 3.45 may be reworked as follows:

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \left(\mathbf{J_r}^\top \mathbf{J_r} + \mu \mathbf{I}\right)^{-1} \mathbf{J_r}^\top \mathbf{r} \left(\boldsymbol{\beta}^{(k)}\right) \tag{3.48}$$

When the combination coefficient is very small, Equation 3.48 approaches Equation 3.45, representing the Gauss-Newton method. However large combination coefficient values will result in Equation 3.48 approximating the steepest descent algorithm which may be mathematically described as follows:

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \alpha \nabla \mathbf{S}(\boldsymbol{\beta}) \tag{3.49}$$

Indeed, for very large values of μ, the learning coefficient in the steepest descent algorithm, α, may be approximated using:

$$\alpha = \frac{1}{\mu} \tag{3.50}$$

3.5.5 Genetic Algorithms

It is not always possible to apply gradient descent algorithms to optimise for a given problem space. In these cases other algorithms are required to search the problem space in an effective and efficient manner. Genetic algorithms are nature-inspired iterative methods commonly used to find solutions to optimisation and search problems. Such systems typically start with a population of randomly seeded candidate solutions and then use a measure of fitness or success to identify the most promising candidates. These identified candidates are then used in the 'breeding' stage to produce the next generation of candidate solutions, relying on nature-inspired operations such as mutation and crossover to define the new solutions properties. This process is then repeated until a suitable solution is found.

In order to use genetic algorithms, the system being optimised must first be defined using a 'gene' that fully specifies all tunable parameters in a single long entry. This gene is used to define each of the candidates and modified during the breeding stages

to produce the next generation of candidate solutions. Both crossover and mutation are shown in Figure 3.23. With crossover, two promising genes are selected at random and mixed together to yield two new child genes. In the case of mutation, small parts of the gene are randomly changed while the majority of the gene is maintained. Children of this process therefore maintain the majority of their one parents genetic information.

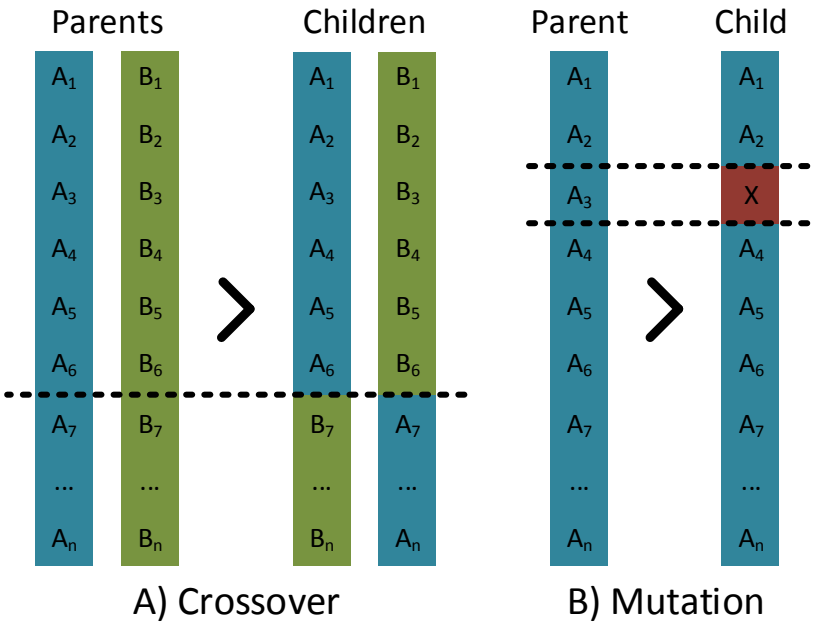


Figure 3.23: Crossover and mutation regimes typically used in the gene breeding steps of genetic algorithms. Crossover results in child genes which contain a combination of parameters from each of the parents genetic data as-is, while mutation results in small, previously untested, modifications to a subset of selected parameters within one parent gene.

Alongside breeding, it is common practice to also include a subset of the most promising genes in the next generation of candidate solutions. Without this pass-through of genetic information it is possible for the solutions to become worse - losing previously good solutions. Some random genes are also added to the next generation to ensure that the gene pool doesn't stagnate or converge onto a single local maxima.

The exact breeding and gene generation methodology is an important factor when designing genetic algorithms. If performed incorrectly, the system can stagnate resulting in little to no change in subsequent generations of candidate solutions. The fitness function or measure of success also plays a critical role in the performance of a given genetic algorithm. If the fitness has been poorly defined the candidates will fail to find an optimal solution.

Genetic algorithms can be very time intensive, especially as the number of free parameters, and in turn gene length, increases. The challenge behind suitably defining a breeding algorithm and fitness function, alongside the large number of weights

requiring optimisation makes this approach less suitable for medium to large scale neural networks.

3.5.6 Particle Swarm Optimisation

Particle swarm optimisation is a stochastic optimisation technique developed by Kennedy and Eberhart in 1995 [80]. It was designed to model social behaviour, such as that seen in flocks of bird or schools of fish. Since its initial conception, particle swarms have been used to optimise a number of different problems, including neural networks. The original model was further improved in 1998 by Shi and Eberhart through the addition of an inertia weighting [81]. This extra weighting typically decreases its influence as the simulation progresses helping the final solutions to settle.

Particle swarm optimisation starts with a random population of points on the problem space, termed particles. Each particle has a velocity, v_n; inertia weight, ω; fitness value, which is found using a pre-defined fitness function; and current position, P_n. On each iteration the position of the best fitness achieved by the particle, P_b, is checked and updated if necessary. The position of the global best fitness achieved by the whole system, P_g, is also stored. These values are then used to calculate a new velocity, v_{n+1} using the following equation:

$$v_{n+1} = \omega v_n + C_1 R_1 (P_b - P_n) + C_2 R_2 (P_g - P_n) \tag{3.51}$$

where C_1 and C_2 are constants selected by the designer, R_1 and R_2 are random numbers of the range $(0, 1)$. Once the new velocity has been calculated, the position is then updated as follows:

$$P_{n+1} = P_n + v_{n+1} \tag{3.52}$$

A maximum velocity is often used to clamp the system and ensure that the target minima are not skipped over by the swarm as a whole.

This algorithm has notable similarity to that of genetic algorithms. Both use an initially random population, performing iterative searches on the problem space using this population and a user defined fitness function. However, particle swarm optimisation does not use evolution or generations of populations, rather the algorithm maintains the initial population and instead moves them around the problem space.

The field of particle swarm optimisation has received significant interest within recent years, with around 700 new publications added per year between 2010 and 2014. With so many new particle swarm optimisation variants and test functions it is impossible

to compare and identify the best variation or method, a problem readily identified by Zhang *et. al.* in their comprehensive and detailed review paper [82].

Particle swarm optimisation techniques suffer from high computational complexity, slow convergence and a considerable sensitivity to parameters. Additionally these methods do not handle the relationship between exploitation (local searches) and exploration (global searches) very well, and they therefore easily converge to local minima. Considerable effort has gone into overcoming these issues [82], however often the additional algorithm complexity only acts to increase the computational cost, further limiting the cases where such methodologies are suitable.

3.6 Conclusions

This chapter has touched upon some of the key concepts in the field of ANNs. From the early single layer binary networks to the modern and deep CNNs, each advancement brings improvements to the power efficiency, training speed or ability to represent complex non-linear problems. Over the last 5 years there has been a considerable quantity of research publications, making it impossible to compare and quantitatively identify a best solution or system. Within this huge range of research efforts, however, there have been some key discoveries that have helped the field move forwards in leaps and bounds.

In this chapter the field of ANNs has been divided into four key considerations: artificial neuron models, activation functions, network topology and network training algorithms. Each of these topics must be reviewed by a designer when developing a new ANN system. Artificial neuron models have seen little advancement in recent years, with the main focus shifting to activation functions and network topologies.

Within the topic of activation functions, ReLU has become the latest and most popular activation function for use in a wide range of applications. Many variants and small modifications have been tested, however the simplicity and effectiveness of the ReLU function makes it likely that this trend will continue for the foreseeable future. Despite ReLUs popularity, the logistic sigmoid function still finds significant use within neural networks with recent work arguing that properly initialised deep sigmoidal networks consistently outperform deep ReLU networks during training [47].

Network topology is arguably the area most influenced by the target application. Modern classification and regression tasks frequently use deep FFNNs, achieving results comparable to human performance on a number of test datasets. Image recognition and object identification tasks require systems that offer shift-invariance, such as CNNs. Their biological inspiration makes them particularly suitable for vision related tasks and it seems likely that future advancements in such systems will be achieved through closing the theoretical gap between biological and CNN structures. Tasks involving

temporal data, such as speech recognition, will often require use of RNNs. These networks contain feedback paths, making them more complex to train than their FFNN counterparts. Of these networks, LSTMs have become popular, with their ability to capture long-term dependencies and avoid the vanishing gradient problem, making them particularly suited for such tasks. Hopfield networks and Boltzmann machines form examples of RNNs, with a fully connected arrangement and unique I/O neurons that act as both input and output for the network. These models are capable of completing partial input patterns, making them particularly suited for classification tasks involving partial, noisy or corrupt input data.

Network training forms the final key element in ANN design. From the original theory of Hebbian learning to modern day training techniques, the questions surrounding biological learning remain largely unanswered. Despite this challenge there have been significant advancements within the field that have allowed neural networks to achieve state of the art results. Back propagation is the earliest and most prolific training algorithm, with its simplicity and intuitive process making it especially suited for classification and regression tasks. Other gradient descent training algorithms have shown considerable promise, however the addition of extra computation requirements often makes such approaches unsuitable for large scale networks. Genetic algorithms provide another nature inspired approach to network training, yet these methods are somewhat unpracticable in their convergence and lean heavily on the designer to accurately specify the fitness functions and breeding algorithms. Particle swarm optimisation has provided optimisation solutions for a number of different problems and shows considerable promise in neural network implementations. As with other optimisation techniques these methods require significant computational resource. It therefore seems likely that future progress within this topic will be heavily driven by advancements in both computational power and efficiency.

Many of these choices depend on, not only the end application, but also the target architecture or hardware. Implementing these systems on generic computing hardware has become an increasing challenge as ANNs have grown from several neurons to several million neurons and billions of synapses. Additionally, training these networks can consume considerable time and computing resource and there is therefore considerable commercial and academic interest in the acceleration of such processes. Google's own Tensor Processing Units, for example, were developed in response to the increasing scale and popularity of neural network applications within their own data centres. These custom hardware solutions realise speed improvements of between $15 - 30\times$ that of contemporary Central Processing Unit (CPU) or Graphical Processing Unit (GPU) stacks, with power improvements of between $30-80\times$, making them a critical component in the continued success and growth of Google's own computational capabilities [83]. These hardware systems are critical in the continued advancement and development of ANN solutions and research. The next chapter in this thesis reviews the key neural network hardware systems available to the research community, comparing their designs and implementation limitations.

Chapter 4

Neuromorphic Hardware

First coined by Carver Mead in 1990 the term 'neuromorphic hardware' is used to describe a circuit inspired by the energy efficiency and processing capabilities of biological neural systems [6]. By this time, the process of combining thousands of transistors into a single Integrated Circuit (IC) was well established - termed Very-Large Scale Integration (VLSI), this process is still used today to build modern processors and ICs. Noting a considerable and fundamental difference in the power per computation between VLSI systems and even insect brains, Mead argued that there was clearly something undiscovered regarding the way that biological neural networks process data. Drawing comparisons between the estimated power used by an individual neuron and that of a single transistor, it has been shown that they yield similar power per activation. With this observation, Mead notes that:

> "The disparity between the efficiency of computation in the nervous system and that in a computer is primarily attributable not to the individual device requirements, but rather to the way the devices are used in the system." - [6]

Alongside this conclusion, two primary causes of energy waste are identified in digital processing systems. Firstly, focusing on the dynamic power consumption, it was observed that a considerable portion of the energy consumed in a VLSI system is used to charge the connections and wires between the transistors. The capacitance of these connections is at best comparable to that of the of gate itself, and often forms the majority of the node capacitance. Secondly, modern day processors make use of many thousands of transistors when performing a single operation. As a result, these operations consume several thousand times the computational power of a single transistor while also increasing the static power consumption due to an increase in leakage currents. Neuromorphic hardware, however, makes use of transistors fundamental non-linearities to perform computation in a manner similar to that of biological neural networks. This greatly reduces the number of underlying building block elements required to perform the computation.

Neuromorphic hardware has come to include any designs built to accelerate or efficiently implement neural simulations and Artificial Neural Networks (ANNs). Despite the considerable advancement of modern processing technologies, simulations large and yet detailed enough to emulate multiple cortical areas are still impractical on standard processing solutions. As a result many hardware alternatives have been explored, aiming to provide more powerful, affordable and efficient neural simulation platforms [84]. Alongside the desire to implement and reproduce the cognitive computation performed by biological nervous systems, there is also the hope that neuromorphic systems will provide clues for a new generation of asynchronous, low-power, parallel computing solutions [14]. These systems stand to bridge the gap in computing power when Moore's law has fully run its course.

Hardware neuron implementations are much more energy efficient than neurons simulated on generic computational devices such as Central Processing Units (CPUs), making them well suited for large scale real-time neural emulation. These hardware systems, however, are often less flexible than software implementations. As a result it is up to the designers of these systems to select suitable models, redundancy and hardware constraints to allow sufficient flexibility for the end applications. The relative merits and correct mix of analogue and digital systems in neuromorphic computing remains an open subject for further research, with the answer likely dependent on the hardware's target application [14]. It is highly unlikely that one absolute solution or implementation will be discovered and instead a range of available hardware systems may be beneficial, allowing the users choice when it comes to model and architecture implementations.

This chapter considers the motivation behind many neuromorphic hardware efforts before reviewing some of the key neuromorphic systems in use and development stages today. These systems are each designed with a clear purpose or target application, offering accelerated and efficient solutions to otherwise challenging implementation problems. These same motivators led to the development of a hardware-optimised activation function that will be developed in Chapter 7 and the novel grid-architecture neuromorphic system introduced in Chapter 8.

4.1 Neuromorphic Hardware Motivation

There are a many different motivators for the development of neuromorphic hardware. These motivations may be divided into two categories, computationally focused design and biologically focused design. While the advantages of neuromorphic hardware are shared between the two, the relevant weighting of different motivations and design requirements often separates each of these approaches. Despite these differences, it is important to note that progress in one category will often yield improvements when applied to the other, such that the two research fields will never be fully independent of one another.

4.1.1 Computationally Focused Design

Computationally focused design includes implementations intended for the application of neural network systems to computational problems, such as image recognition or speech generation. These systems are typically less focused on biological accuracy and instead require large numbers of neurons, often arranged into layered networks. The underlying design and implementation of such computationally focused systems becomes increasingly important as machine learning and deep learning find wider application in commercial and industrial solutions.

Energy Consumption

As with many computational systems, the energy consumption per compute is an important metric when it comes to neuromorphic hardware design. The ever increasing scale of neural network implementations means that small improvements to the efficiency of neuron models can have compound effects on the overall network efficiency. An improvement of just 10nW per neuron, for example, would save a total of 10W on a modern network of 1 billion neurons. When compared against traditional simulation efforts, neuromorphic hardware is far more energy efficient [84], a trend reflected in the observed power consumption of modern-day CPUs when compared against Application Specific Integrated Circuit (ASIC) or even Field Programmable Gate Array (FPGA) solutions.

Speed of Computation

Another common metric when considering computational systems is that of computation speed. Often this must be weighed against the physical system size and energy consumption, where improvements in one area frequently results in compromises in the others. While some systems are capped at real-time $1:1$ speeds (when compared with biological systems), it is more common to aim for accelerated computation, where the system operates at faster than biological real-time. In these cases, systems that require real-time operation may use time-division multiplexing techniques to simulate many neurons with a single hardware neuron element. This technique allows systems to emulate much larger networks than they would otherwise be capable of.

Network Scaling

The scale of the networks that may be emulated using a neuromorphic system is also an important consideration when designing and selecting suitable hardware implementations. Efforts to emulate multiple cortical areas require networks of considerable scale. Alongside this biological modelling work, Deep Neural Networks

(DNNs) have proven to be capable of relatively complex classification tasks. As seen in Chapter 3, each additional layer provides new abstraction to the data processing. As greater computation complexity is required from these networks, the scale of these networks will therefore also increase. This trend has been evident in historic research efforts within the field and will likely continue for the foreseeable future.

Despite the drive for ever increasing network size, there is also substantial motivation for designing smaller network hardware which will support scaling in an efficient and practical manner. These systems are often intended for mobile applications, distributed data processing systems, or Internet of Things (IoT) devices. In these cases the aim is to achieve good resource utilisation, ensuring that the network scale is sufficient for the intended application. Overspecifying the network size in these instances results in redundant resource and often equates to wasted fabrication space or energy. In these applications, neuromorphic hardware offers a compact and efficient solution.

Cognitive Computation

Neuromorphic systems can easily be mistaken for a subclass of Turing complete machines. Indeed some neuromorphic systems, termed *neural Turing machines*, represent the full realisation of this association. These neural Turing machines form an abstraction of Long Short-Term Memory (LSTM) networks and result in a neural network that may be trained using gradient descent algorithms, yet at the same time yields an insight into the networks inner workings [85]. It is important to recognise that, despite this exception and the apparent similarities between neuromorphic systems and traditional Turing machines, cognitive processing is very different to Turing computation. A cognitive computational system may provide intuition or be capable of leaps in logic otherwise unattainable by standard processing technologies. This emulation of thought, however, comes at the cost of precision and uncertainty, resulting in systems that can provide inspired yet somewhat uncertain solutions to a task.

An example of this difference can be observed when comparing the human mind to that of a traditional computer. While a computer can accurately and rapidly provide the answer to an arithmetic problem, it cannot interpret the underlying value or meaning of that answer. The human brain, on the other hand, will often be able to provide an approximate answer to an arithmetic problem alongside an understanding of the underlying meaning. Alternatively, a young child can identify between a cat and dog with apparent ease, while for computers such identification proved a surprisingly challenging task for standard computational techniques. Despite its speed and precision, leaps in logic (including those easily performed by young children) and applying previously found shortcuts to new problems are two aspects of human cognition that remain totally impossible to the computer.

4.1.2 Biologically Focused Design

Unlike the computationally focused designs, biologically focused design is predominantly used when accurate modelling of biological function is required. In such systems energy consumption, speed and scaling are still important factors, however biophysical accuracy becomes the key requirement overriding these otherwise desirable properties. The complicated non-linear relationships and circular dependencies underlying the models often means that only small numbers of neurons may be simulated, even when using modern processing solutions. As seen in Chapters 2, there are trade-offs that may be made to make this modelling task easier, however in some cases the loss of biophysical accuracy is unacceptable. In such situations neuromorphic hardware offers a possible solution, utilising the non-linearities in transistors and other analogue design techniques to provide efficient, fast and accurate neuron models. Alongside producing accurate neuron models for experiments, these systems may one day offer a practical method for interfacing technology with biology, revolutionising the fields of both prosthetic design and paraplegic surgery.

The biological nervous system's ability to learn and adapt also holds considerable promise if successfully replicated in technology. This could provide systems capable of advanced noise rejection, fault tolerance and on-line learning and improvement. While achieving human-like learning and understanding is, at present, beyond what is technically feasible, it is often hoped that systems demonstrating learning capabilities will come to solve previously unsolved problems.

4.1.3 Summary

There are many different motivations when designing neuromorphic hardware. While most are focused on what improvements the current technology may offer today, it is the promise of tomorrows solutions that is often more exciting and eagerly anticipated. At present, almost all neuromorphic systems in development are designed with both power and speed in mind. Often these designs are made in the hope that enabling larger networks to be emulated will either unlock previously hidden understandings behind the function of the biological nervous system or provide human-like performance on some given tasks.

With the motivations for neuromorphic systems identified, this chapter now discusses five key neuromorphic systems used in the acceleration of neural network applications. These systems all differ in their underlying structure and design, reflecting their different target applications.

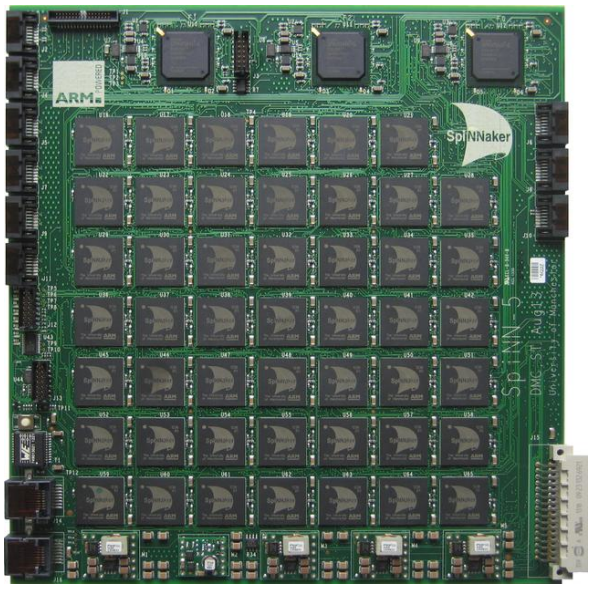


Figure 4.1: A SpiNNaker board, with its 48 'processing nodes' or ICs. These boards are attached together by means of a packet-switched network, allowing the system to scale, supporting millions of neurons. Image taken from [88].

4.2 SpiNNaker

SpiNNaker is a general purpose real-time simulation system for large-scale spiking neural systems of up to 1 billion neurons, first detailed in 2006 by Furber, Temple and Brown [86, 87]. Unlike other neuromorphic designs, SpiNNaker is functionally a hardware-accelerated software simulator that focuses on flexibility, power-efficiency and fault-tolerance. This software implementation means that there is greater abstraction from the underlying system itself, providing considerable flexibility when selecting a suitable neuron model. Unlike other parallel computer architectures, SpiNNaker builds on the assumption that tasks easily split into an arbitrary number of threads (like that of neural network modelling) will perform better on systems supporting large numbers of low-area low-power processors, rather than the typically larger optimised performance processors [86]. This largely reflects the pattern seen in biology, where large numbers of simple computational units (neurons) perform complex tasks with significant energy efficiency.

The SpiNNaker system is formed from a network of circuit boards (like the one shown in Figure 4.1), each supporting an array of identical custom ASICs termed 'processing nodes'. Each processing node contains two ICs: the chip multiprocessor and SDRAM. The chip multiprocessor IC contains 18 ARM968 processors situated in synchronous islands, while the SDRAM IC provides the memory used to store the synaptic data. The processors share access to the SDRAM using self-timed packet-switched Network-on-Chip (NoC), built using the CHAIN technology [89].

The synchronous processor islands are surrounded by a light-weight, packet-switched

asynchronous communications infrastructure [90]. An address-event signalling NoC fabric is used to form an efficient multicast communications system, providing the interconnectivity between the different chips. In place of the usual bus-based fabric, a second self-timed packet-switched fabric is used to carry the signals, decoupling the clock domains within and across the chip multiprocessor IC. All on-chip and off-chip message sources are merged into a single stream of packets using an asynchronous arbiter situated at the ICs interface [87]. A multicast router is then used to direct incoming packets using the source ID and a routing Look-Up Table (LUT).

One of the 18 processors is selected during the boot process to act as a 'monitor processor', dedicated to the system management functions rather than the normal neural modelling operations. The remaining 17 processors provide the neuron modelling solutions and are termed the 'fascicle processors'. This flexible boot-up assignment removes a major single point of failure, ensuring that the system has greater fault tolerance.

SpiNNaker is a fully digital solution and uses integer values rather than floating point arithmetic. Each fascicle processor can support the simulation of about $1,000$ Integrate and Fire (IF) neurons, where each neuron has $1,000$ inputs and an average firing rate of 10Hz [86]. The final system is designed to scale from single chip operation to implementations involving tens of thousands of chips [91]. Action Potentials (APs) are treated as all-or-nothing events and it is therefore assumed that the information is only carried in the spike source and spike timing data.

The packet-switched network and address-event representation helps decouple the implemented network topologies from the physical chip architectures. This means that arbitrary network topologies may be easily tested, without requiring modification to the underlying technology. The SpiNNaker communications network is arranged into a torus structure in effort to reduce the maximum separation between any two chips within the network. Alongside their normal function, the processing nodes also provide system level debugging and management operations [92]. With the ability to simulate practically any neural model, and a highly abstracted communications implementation, SpiNNaker offers an extensive and flexible interconnectivity.

Some of the key limitations of the current SpiNNaker system are found in the early design decisions for the system itself. Lack of support for native floating-point operations greatly limits the implemented networks and constrains the range and fidelity of any neural models used. Additionally, loading and reading data in SpiNNaker is a time consuming task due to bottlenecks in the communications infrastructure. This makes rapid iteration challenging to implement, with considerable downtime if the network requires off-board modification between runs. The use of standard ARM968 processors offers good power efficiency, however these processors are not optimised for neuromorphic operations meaning that they fall short of the energy efficiencies achievable had the system been designed from the ground-up.

Using quoted figures for the power-efficiency of ARM968 processors it was estimated

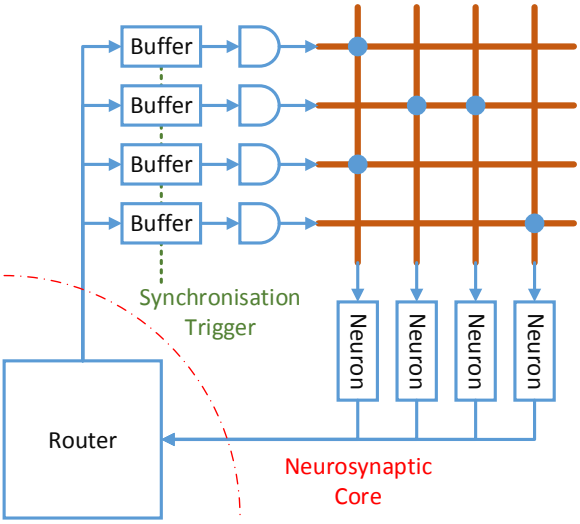


Figure 4.2: Structure of the TrueNorth neurosynaptic cores, shown connected to a local router. The router provides connection to the other neurosynaptic cores, located both on- and off-chip.

that a neuron with 1,000 inputs firing at a mean rate of 10Hz will consume $23-36\mu$W of power [86]. This relatively closely agrees with experimental results, where a simulation of 17,000 real-time neurons requires around $0.55-0.75$W in total [93], giving a measured power consumption of approximately $32-44\mu$W per neuron.

4.3 TrueNorth

TrueNorth was developed by IBM as part of the Defence Advanced Research Projects Agency (DARPA) SyNAPSE project. This neuromorphic system was designed under 7 key principles: minimising active power, minimising static power, maximising parallelism, real-time operation, scalability, defect tolerance and hardware-software one-to-one equivalence [94]. Much like SpiNNaker, TrueNorth is a fully digital and hugely parallel system. Unlike SpiNNaker, however, TrueNorth forms a non-von Neumann architecture resulting in a biologically inspired arrangement of mixed memory and processing elements within the silicon. Each IC provides an aggregate of 1 million neurons and 256 million synapses and with demonstrations of 1, 4 and 16-chip systems, this hardware provides an effective and scalable solution [94].

Each of the TrueNorth ICs contain a 2D array of 4096 neurosynaptic cores. Shown in Figure 4.2, these neurosynaptic cores contain a mix of neurons and memory, used to store the neuron connectivity information and parameters. Localising the data storage in this way provides a significant advantage in energy efficiency, greatly reducing the communications overhead.

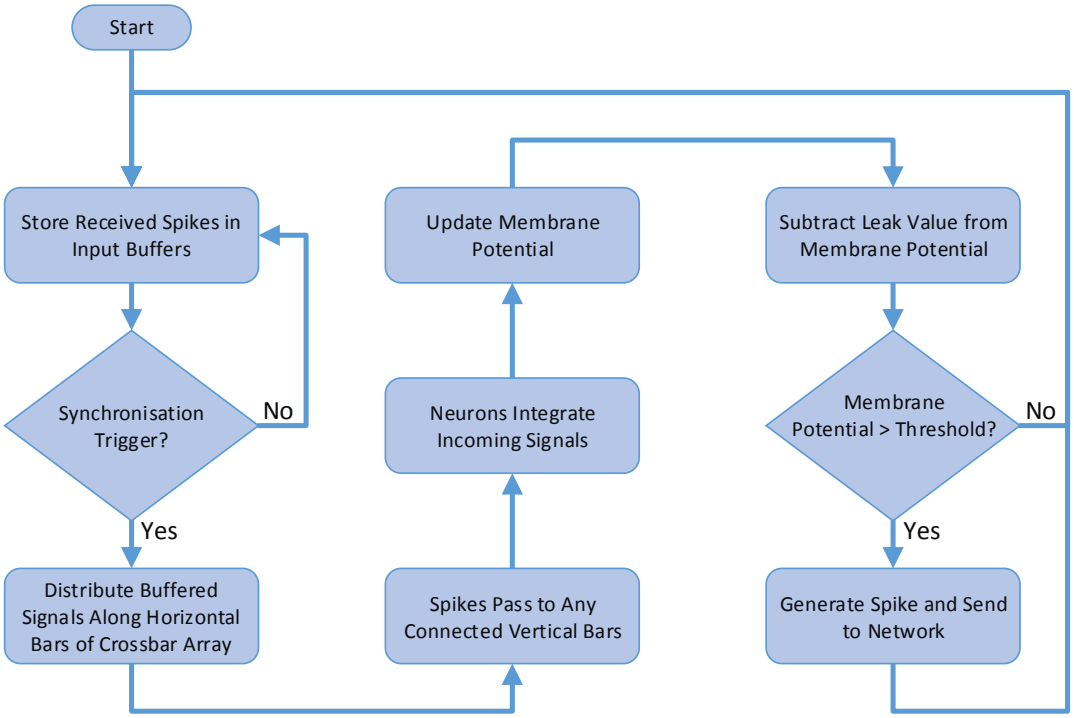


Figure 4.3: Tasks performed by the neurosynaptic core during a single time step. Any received signals that arrive after the synchronisation trigger will be stored in the buffers ready for the next cycle.

Each core implements 256 neurons and 64,000 synapses. The synapses are arranged into a synaptic crossbar array, as shown in Figure 4.2. Buffers are used on the crossbar inputs to ensure that computation is synchronous and deterministic. This deterministic nature of the TrueNorth architecture allowed software simulations to be developed with one-to-one hardware-software equivalence accelerating development and verification tasks [95].

The neuron model used within each neurosynaptic core is a modified leaky IF model, detailed thoroughly by Cassidy *et. al.* [96]. This model has been used to demonstrate all twenty of the fundamental neurocomputational properties of biological neurons identified by Izhikevich [21]. In each system time-step, the neurosynaptic cores must buffer all incoming inputs. On receiving a synchronisation trigger these buffered inputs are then passed through the crossbar array before being used to calculate the new neuron membrane potentials and generate new spikes. This full cycle is shown in Figure 4.3, and must be fully complemented in a single system time step.

TrueNorth has a quoted typical power consumption of 65mW making it highly efficient when compared against other neuromorphic systems [94]. Cassidy *et. al.* also found the system capable of providing 46 giga-synaptic Operations Per Second (OPS) per watt when running a complex Recurrent Neural Network (RNN) with a $20Hz$ average firing rate and 128 active synapses per neuron in real time [97].

4.4 Neurogrid

Neurogrid, developed at Stanford University, is a complete hardware neuromorphic solution. Using a mixed analogue-digital system, this energy-efficient neuromorphic implementation targets large-scale biological real-time neural simulations [98].

The system is constructed from 16 identical ASICs, termed Neurocores. These Neurocores are arranged into a binary tree routing network, where 12-bit packets are used to provide core-to-core communications [99]. These packets may be passed in either direct mode, where the message travels in a provided direction through the tree; or split mode, where the message is copied to both children of the current node.

Each Neurocore contains a 256×256 silicon-neuron array, a transmitter, a receiver, a router and two Random Access Memory (RAM) regions. The circuits used within the neuron array may be divided into 10 distinct segments according to their function. These are the soma and dendrite circuits, as well as four gating-variable and four synapse-population circuits. The design and models behind each of these circuits is provided in considerable depth by Banjamin *et. al.* [98]. Additionally, optional local dendritic connections known as arbors may be used to distribute the synaptic inputs between spatially neighbouring neurons.

A real-time network of 983,040 neurons and roughly eight-billion synapses was generated to simulate a recurrent inhibitory network with a total of 15 layers. Each layer was mapped onto a different Neurocore. During this simulation Neurogrid was found to consume 2.7W of power [98]. A further study showed that an equivalent implementation using arbitrary connections would consume an extra 0.4W to route the spikes [100]. This results in a total power consumption of 3.1W, yielding an efficiency of 941pJ per synaptic activation.

Neurogrid does not, by default, support synaptic plasticity due to an underlying architectural decision in which spatially neighbouring neurons share the same spatially decayed input. In this way the system may be considered to have a shared dendrite structure. The addition of an FPGA daughter board allows for the implementation of long-range connections and individual connection weights. These individual connections weights may then be used to support spike-timing-dependent plasticity [100].

4.5 Tensor Processing Unit

As arguably the first commercially focused neuromorphic hardware, the Tensor Processing Unit (TPU) was originally developed by Google in 2013 to address a computing resource crisis identified within Google's own data centres. Google recognised a growing number of users utilising features dependent on machine learning tasks. The use of excess CPU and Graphical Processing Unit (GPU) computing power

was no longer a cost effective method to deliver for these tasks and it therefore became a priority for Google to develop a new dedicated low-cost solution. As such, the TPU is an ASIC designed to accelerate neural network machine learning tasks, with a specific focus on supporting Google's own machine learning language, known as TensorFlow. The original solution was deployed in Google's data centres in 2015 and while these chips are not available for purchase, it is possible to rent computing time using them through a Google provided service.

The architectural information on these chips is somewhat limited due to the commercial interests surrounding this product. In 2017, Google published a handful of articles detailing the design and development of the original TPU systems, the following information is therefore applicable to TPU v1.0 and further improvements in power efficiency and speed may be expected in later models.

The TPU v1.0 was originally designed to run DNN inference tasks $10 - 30\times$ faster with $30 - 80\times$ better energy efficiency (when compared with contemporary CPUs and GPUs on the same technology) [101]. This improvement is largely down to the underlying 2D vector support that greatly expands the operations per instruction, allowing instructions to operate on a 256×256 array of 8-bit data. This 256 wide vector operation, however, means that all internal elements must be connected using 256-byte-wide paths.

At the heart of the TPU is the matrix multiply unit that contains 256×256 8-bit multiply-accumulators. A set of $4,096$ 256-element 32-bit accumulators is used to produce the 16-bit products from this matrix. This multiply unit calculates one full 256-element partial sum per cycle. The weights are stored using on-chip First-In, First-Out (FIFO) registers that read from an off-chip 8 Gibibyte (GiB) DRAM. Both of these memories have error detection and correction hardware built-in to extend the products lifetime.

The TPU was designed to be a co-processor on the Peripheral Component Interconnect Express (PCIe) I/O bus, allowing it to plug into existing servers using the same technology as GPUs. The system is therefore designed to run whole inference models on-board the TPU, reducing the I/O requirements. Unlike GPUs, the TPU requires the host server to send instructions over the PCIe bus rather than sequencing itself, resulting in a solution which bears closer resemblance to that of a floating-point coprocessor. To reduce bandwidth dependencies the system uses Complex Instruction Set Computer (CISC) instructions and decoupled-access/execute memory instructions.

While new versions of the TPU have been released, published information on the architectural improvements has yet to be announced. Some ideas on future improvements of the TPU v1.0 were provided by Jouppi *et. al.* [83]. The development team identified the short development cycle for this system (roughly 15 months from conception to installation) as a limiting factor, stating that more aggressive logic synthesis and block design could increase the clock rate by 50%. Additionally, using GDDR5 memory would increase the weight memory bandwidth by more than $5\times$.

With these relatively simple changes identified it is therefore possible to see how TPU v3.0 represents a significant improvement over it's predecessors.

The lack of published information and constrained availability of the TPU architecture represents the largest limitations for this system. Without published scales and power consumptions it is hard to compare the system with other existing neuromorphic systems. Equally, the lessons learnt during the design of the TPU architecture are unlikely to directly benefit the wider neuromorphic community due to the corporate interests associated with this project.

4.6 Loihi

Loihi is another fully-digital neuromorphic chip, developed by Intel on their 14-nm process [102]. As with many other neuromorphic systems, this chip is designed to implement spiking neural networks in an efficient and effective manner. Each chip contains 128 neuromorphic cores, three embedded x86 processor cores and off-chip communication interfaces. An NoC is used to pass communications packets between cores, while encapsulated packets are used for off-chip communications in four planar directions. These protocols support scaling the system to 4096 on-chip cores and a full 16,384 chip array.

The neuromorphic cores make use of a fixed-size discrete time-step leaky integrate-and-fire neural model. Each core implements 1,024 primitive spiking neural units or compartments, which are grouped into sets of trees constituting neurons. This yields a neuron density of 2184 neurons per mm^2. Unlike other neuromorphic systems discussed, each core includes a programmable learning engine that may be used to modify the synaptic state variables as a function of historical spike activity. These learning rules are programmed using microcode making them more flexible than other fixed learning system implementations. The Loihi system may be programmed using a Python API, as demonstrated by Lin *et. al.* using the MNIST benchmark dataset [103].

4.7 System Comparisons

The recently developed neuromorphic systems have all been designed to simulate large numbers of neurons, with efforts like SpiNNaker and TrueNorth utilising inherently scalable technologies to support huge neural networks with several billion synapses. Since these research efforts have different target applications and goals, it can be difficult to provide meaningful comparisons between them all. Table 4.1 provides a quantitative comparison between each of the three most publicised research efforts. From this table it may be seen that TrueNorth offers the best power density, an important factor when considering the overall power consumption of the system as it is

Table 4.1: Quantitative comparisons of TrueNorth, Neurogrid and SpiNNaker, showing that TrueNorth yields the best power density while SpiNNaker's highly flexible software implementation results in a high power consumption [104].

System	Neuron Count	Synapse Count	Power Density (mW/cm^2)
TrueNorth	$16,000,000$	$4,000,000,000$	20
Neurogrid	$1,000,000$	$8,000,000,000$	50
SpiNNaker	$20,000,000$	$20,000,000,000$	$1,000$

scaled. This said, SpiNNaker supports the largest networks, and along with its software implementation, it offers the most flexible and reconfigurable system available. These features come at high cost when considering the power density, with the SpiNNaker system using around $20\times$ more power per cm^2. Neurogrid's neuron count is relatively small when compared against the other two systems, however this systems supports many more synapse connections per neuron. This high ratio of synapses means that this system can simulate dense network structures, allowing researchers to push the limits on network interconnectivity. Despite the extra connectivity provided, Neurogrid's analogue elements mean that it only uses about $2\times$ more power per cm^2 when compared to that of TrueNorth.

Unlike the first three systems discussed, Google's TPU was designed as an internal product and sold as processing time on a neuromorphic system. There is only limited information available for this system, due largely to the commercial interests associated with its business model. Despite the lack of information, it appears that this systems operates more like a hardware accelerator for neural network learning and inference tasks. A key limitation is that these tasks must be developed using the TensorFlow language. Google has now produced three incremental versions of this system and reports considerable (but un-quoted) energy savings within their own data centres as a direct result of the systems installation.

Intel's Loihi system is a relatively new development, which includes its own on-chip learning engine. This system stands to benefit from Intel's close relationships with chip fabrication facilities and has so far been implemented on their own 14-nm process. Loihi's maximum neuron density of 2184 neurons per mm^2 is marginally worse than TrueNorth's, however this may be argued as a justifiable cost for the Loihi's expanded feature set.

4.8 GPUs and Other Accelerators

There are other solutions, besides neuromorphic systems, that are used to achieve accelerated neural network implementations. While GPUs are not neuromorphic hardware, they can offer considerable acceleration in ANN applications when compared

against standard CPU solutions. Neural network inference and training are hugely parallel tasks, with each node performing operations based only on its own input data and model properties. As a result GPUs, which are optimised for large parallel operations, perform ANN tasks very effectively.

Spiking neural networks, however, do not map so well to GPU architectures. This is due to the networks dependence on its internal interconnectivity, meaning that a large bandwidth between the processing elements is required. Spiking neural networks are also designed to capitalise on their sparse nature of operation. It is the sparse APs of spiking neural networks that yields high energy efficiency when compared with other ANNs. Optimising a system for these two properties often requires new custom hardware, making GPUs impractical for such applications.

Other acceleration approaches may also be taken to improve neural network performance. Custom instructions can be added to processors to provide designers with highly efficient and rapid operations, such as the calculation of the reciprocal of a number. These custom instructions complement existing processor technologies, improving the performance by committing extra chip area to efficiently or rapidly solving a small but regular part of the larger task. This custom instruction approach provides further promise when achieved through flexible reconfigurable processing fabric located alongside the processor. In the context of Intel's recent acquisition of Altera, alongside mention of Altera FPGA co-processors, it is highly likely that this approach for task acceleration will become more mainstream, making custom instruction design an important part of function acceleration.

4.9 Conclusions

Neuromorphic systems are a fundamental part of neurological research and simulation tasks. With correct application, these systems have the potential to provide processing solutions that go beyond Moore's law, enabling designers to continue the advancing trend in computational power. Existing neuromorphic systems already offer solutions to previously challenging computational tasks, helping to close the gap between computational processing and human cognition. At the same time, they are providing significant insight into the inner function of biological nervous systems, enabling a wide variety of research. Leveraging function acceleration and optimisation, these systems are constantly growing in scale and processing potential. The main neuromorphic systems in development today are very different when compared against the minimal and analogue systems designed by Carver Mead when he first coined the term. The power efficiency of natural neural networks, however, has continued to inspire the research efforts within the field. This has yielded a number of implementations capable of better-than-CPU/GPU performance when implementing neural network applications.

Spiking neural networks have become a core focus for many neuromorphic systems. This is partly because such systems may be easily implemented on multi-cast network-based hardware. The fact that these systems are both sparse and event-driven helps realise considerably low power solutions, however this generic communications infrastructure is still very different to the biological nervous systems own interconnectivity.

Other large processor developers, such as Nvidia, have also announced that they are designing new neuromorphic systems. It is therefore clear this field will grow ever more contested, especially as customers depend on applications that utilise neural networks with increasing frequency. Despite the considerable investment that has already gone into neuromorphic research there are still many unanswered questions. The answers to these questions may play a critical role in finding the most efficient and practical implementations of neuromorphic systems. It is therefore important that they are tackled in future systems to ensure that what may otherwise become widely held assumptions do not cause the field to fall short of it's biological inspirations.

Chapter 5

Analogue Hardware Neuron Models

Chapters 3 and 4 introduced two key areas that are strongly inspired by neuronal physiology. Many of these systems have been designed with large scale networks in mind, frequently driven by one of two desires: to produce novel and powerful cognitive computation engines; or to produce large scale simulation systems capable of shedding new light on the otherwise limited understanding of neural function. The neuromorphic systems identified in Chapter 4 utilise a number of digital network principles, such as packet switched networks, to provide effective scalability and support the large number of neurons targeted by the hardware. The neuron models used in these systems are approximations of the behaviour seen in real neurons, and these approximations are carefully selected to ensure that they do not compromise the experimental results. These approximations are often sufficient for the large scale digital stimulation of deep networks that is usually performed on such hardware. However, when a higher level of biophysical accuracy is required these approximations become insufficient.

The development of medical devices that seek to replicate or modulate biological neurons is one area that often requires a higher level of biophysical accuracy. Such devices must be small and energy efficient, and in cases where interfacing with biology is intended, the analogue nature of biological function must also be replicated. These systems often require far smaller networks of neurons than their computational counterparts, emulating biological features such as the Central Pattern Generator (CPG) found at the heart of many rhythmic and repetitive bodily functions. Analogue circuits are particularly suitable for these tasks due to the small network size, analogue small-signal interfacing requirements and low energy constraints.

Alongside medical devices, these highly efficient analogue neurons are sometimes also implemented in mixed-signal solutions, as in the case of Neurogrid introduced in Chapter 4 [98]. These mixed signal systems allow designers to produce solutions that easily interface with standard digital devices while utilising the complex non-linearities

more readily produced in analogue designs. The combination of these two elements can produce smaller and more efficient biophysically accurate solutions, however noise, quantization and signal-conversion becomes a critical design challenge. As well as Application Specific Integrated Circuit (ASIC) implementations, programmable analogue arrays, such as PAnDA, may be used to develop and implement new mixed-signal neural networks [105]. As Moore's law draws to a close these analogue-digital hybrids are likely to find wider application, with the need for continued improvements in processing power helping to offset the higher costs and challenges typically associated with such designs.

There exists a wide range of analogue neuron models within neuromorphic literature today. Indiveri *et. al.* produced a thorough and detailed review of these models, separating them according to their function or purpose [14]. These designs have found use in small neuromorphic chips (such as Neurogrid) as well as custom one-off circuits where efficient neuron dynamics were required.

This chapter introduces some analogue neuron designs and demonstrates some of the fundamental design issues that must be overcome if they are to see wider adoption. The designs in this chapter are targeted towards fit-and-forget bio-electronic implants. These implants utilise artificial neural CPGs to provide functions that can respond to multiple physiological feedback sources. One example application is found in cardiac pacemakers, providing a means for heart rate control that is driven by the respiratory system, ensuring increased blood flow when greater breathing rates and volumes are detected. In a healthy person, this cardiac synchronisation ensures that oxygen reaches the body as required when performing strenuous activities. Modern pacemakers, however, cannot offer this synchronisation - a limitation that leads to many patients reporting a feeling of light-headedness when performing physical activity. The development of a novel artificial CPG to control the pacemakers rate would help close this gap, improving the quality of life for patients. While not explored in this work, these same designs could be implemented in a mixed-signal network to allow researchers to investigate the inner functions of biologically plausible neural networks.

This chapter starts by introducing a full custom analogue synapse built for a $0.35\mu m$ Complementary Metal-Oxide-Semiconductor (CMOS) process. The design and results from this work are detailed in Sections 5.1 and 5.1.2. Section 5.2 introduces a project that investigates the application of analogue neurons in fit-and-forget bio-electronic implants. The neuron model selected for this work is outlined in Section 5.2.1 before the design and validation methods are described in Section 5.2.2. There is considerable interest in the impact of medical procedures, such as Magnetic Resonance Imaging (MRI), on the operation of bio-electronic implants. The Integrated Circuit (IC) was therefore tested in a 3T static field matching the conditions of a standard MRI scanner, as discussed in Section 5.2.3, before the key strengths and issues behind analogue neuron design are discussed in Section 5.3.

Collaborator Contribution

The inspiration for the cardiac CPG design comes from Prof Alain Nogaret who is an expert in emergent synchronisation in local neural networks at the University of Bath. The layout of the CPG IC was performed by Prof John Taylor, also of the University of Bath, who is an expert in analogue IC design. The high field magnet tests were kindly enabled by Hugh Blakes of Siemens MR Magnet Technology at Oxford. All other aspects within the chapter are the work of the author.

5.1 Initial Analogue Neuron Design

In order to explore the potential for analogue neural designs, an initial design was produced using some spare space and I/O on a custom IC fabrication run. Due to external deadlines, the design and layout had to be finished within two weeks, however there was no cost associated with using the spare resource. This was the authors first time designing a custom analogue IC and even with this short deadline, it formed a valuable opportunity to gain insight into both analogue IC design and neuron modelling considerations. The IC used the $0.35\mu m$ CMOS process provided by AMS AG (formally Austria Micro Systems), with a total floorspace of 1mm^2 available for the design.

To maximise the chance of success, and due to the relative inexperience of the author, a pre-existing design was selected from literature. This permitted the chip layout and verification process to become the main focus, whilst also providing a test chip to compare future designs against. This section briefly details this work and considers the resulting outcome and findings, identifying the factors that impact on the design of new analogue neurons.

5.1.1 The DPI Synapse Circuit

One of the the fundamental building blocks of neural function is the synapse, which may be modelled by dedicated synapse circuits. Synapse circuits convert pre-synaptic voltage pulses into shaped post-synaptic currents. These circuits are often used to shape and weight neural inputs in analogue designs. These designs commonly use sub-threshold dynamics (meaning the transistors operate below their threshold region) to translate the pre-synaptic pulses into the longer-lasting post-synaptic currents. A form of integration is also used within these circuits, resulting in the summation of any successive input signals. In effect, these circuits act as custom low-pass filters, with transistors sized to ensure that the resulting current is of a similar shape to that of a biological synapse output.

The circuit chosen for implementation was the Differential Pair Integrator (DPI) synapse

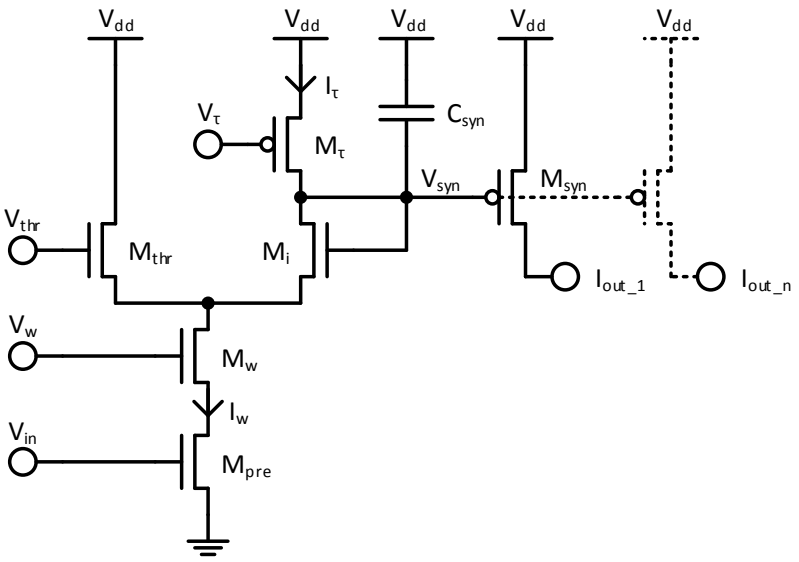


Figure 5.1: The "DPI" circuit, originally defined by Bartolozzi and Indiveri [106] and modified here to provide n parallel outputs using coupled p-FETs. This circuit takes an input square pulse V_{in} and converts it to a post-synaptic Action Potential (AP) shaped signal, performing linear integration and low pass filtering on successive incoming signals.

circuit, originally developed in 2007 by Bartolozzi and Indiveri [106]. This circuit builds upon work by Merolla and Boahen [107] who proposed a log-domain integrator synapse circuit. Merolla and Boahen's design exploited the logarithmic relationship between subthreshold MOSFET gate-to-source voltages and their channel currents, yielding a true linear integrator that was capable of summing and shaping input pulses.

Unlike the original DPI circuit, the design was modified to support synapse weighting through the addition of multiple mirrored outputs (dashed part of Figure 5.1). These extra output points mean that this circuit may be used in a simple binary weighting scheme, where multiple outputs are summed together at the input of the next neuron to generate a weighted input. Since this circuit operates as a current model, these output addition operations may be achieved by simply connecting the desired number of outputs together. This modification makes the design unique from previously published synapse circuits.

Bartolozzi and Indiveri originally define the output of the circuit using Equation 5.1:

$$I_{out}(t) = \begin{cases} \frac{I_{gain} I_w}{I_\tau} \left(1 - e^{-\left(\frac{t-t_i^-}{\tau}\right)}\right) + I_{out}^- e^{-\left(\frac{t-t_i^-}{\tau}\right)} & \text{(Charge phase)} \\ I_{out}^+ e^{-\left(\frac{t-t_i^+}{\tau}\right)} & \text{(Discharge phase)} \end{cases} \tag{5.1}$$

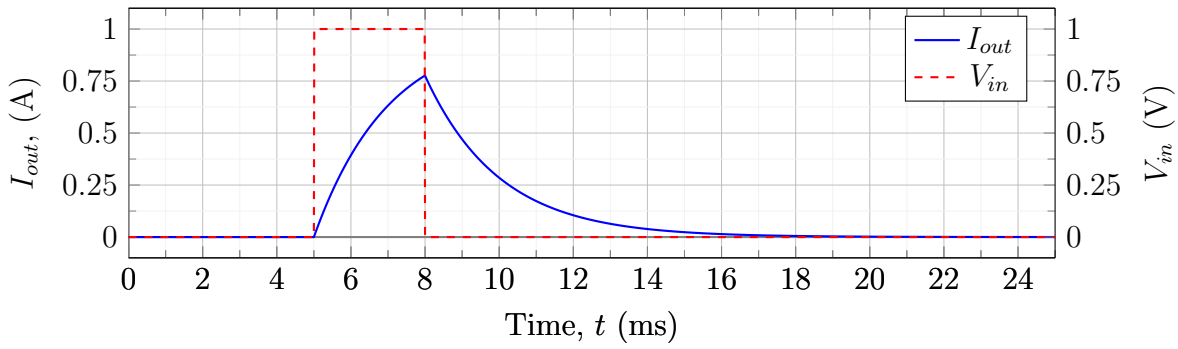


Figure 5.2: Output current for an example DPI implementation. Weights were selected to produce the circuit characteristic curve with a system gain of approximately 0.75.

where I_{out} represents the synapse output current; I_w is a synaptic weight bias current set by the synaptic weight bias V_w; τ is the synapse time constant and I_τ is the time constant bias set by the parameter V_τ; t is the time, where a spike arrives at t_i^- and ends at t_i^+; and I_{out}^- and I_{out}^+ represents the output bias at $t = t_i^-$ and $t = t_i^+$ respectively. I_{gain} represents a virtual p-type subthreshold current defined by

$$I_{gain} = I_0 e^{-\frac{k}{U_T}(V_{thr} - V_{dd})} \tag{5.2}$$

where k is the subthreshold slope factor and U_T is the thermal voltage of the virtual p-FET.

By careful selection of these parameter values, an example DPI output may be generated, as shown in Figure 5.2. The charge and discharge timing and signal size is controlled though the parameters by a combination of sizing of transistor scales and bias voltages. Figure 5.3 illustrates the layout of the DPI circuit on the test chip. A total of 8 p-FETs were used on the output side of this DPI circuit implementation, seen as the 8 transistors located to the far right-hand-side of the layout. All outputs and inputs were broken out to pins on the chip itself to allow complete testing of the circuit using different bias voltages.

With the DPI circuit completed, the final circuit block was sent off for integration into the higher-level IC layout where the design was dropped onto available floorspace and routed to pad-rings around the IC fabric. The design used spare space on another teams IC run and, as a result, the pad-ring and final positioning layout was not performed by the author.

5.1.2 Testing

Once the devices had been fabricated and packaged they were returned for testing. Sadly, it was discovered that the final layout process had misaligned the design block,

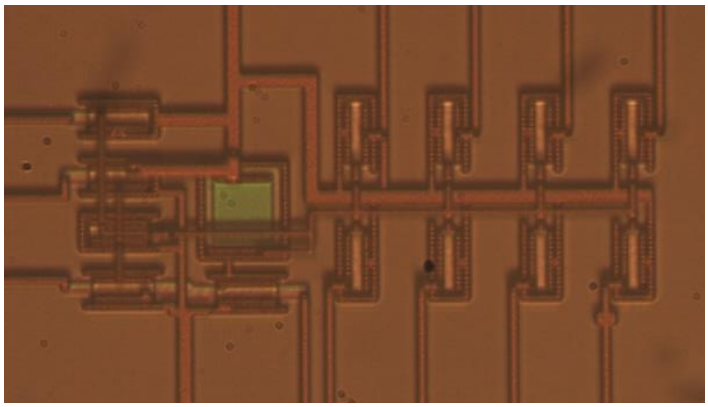


Figure 5.3: DPI Implementation on the $0.35\mu m$ AMS process. The eight output p-FETs may be seen on the right-hand-side of the layout, providing a means for simple output weighting.

resulting in a short circuit to ground of the input for the DPI circuit. This highlighted a number of key issues with the current design pipeline used for this analogue IC process. First and foremost, the Layout Versus Schematic (LVS) and Design Rule Checking (DRC) validation steps were all performed by each designer at the individual circuit block levels prior to the final aggregation and layout process. Had an LVS validation been performed on the final design it would have immediately identified the short-circuit prior to fabrication.

A number of methods for removing the short were considered, including laser ablation. The design and intended test setup were simulated using Cadence Virtuoso to try and find ways of correcting the short circuit, but unfortunately no viable methods could be found. Further simulation work also highlighted some issues with the sizing of the transistors when the full range of process parameter variation was considered. The analogue design process is significantly more sensitive to the placement and sizing of components than its digital counterpart. Analogue designs therefore require significant development time and background understanding. Analogue IC design is frequently considered an art form - with each designer applying their own style and technique. On top of this, the underlying process and technologies used in the fabrication of the chip has a direct impact on the design itself. This means that circuits developed for one fabrication process must often undergo a complete re-design when moving to new scales or technologies. Differences in rail voltages, available floorspace and surrounding noise sources mean that designs must sometimes even undergo a re-design when implemented on the same process and technology. It is this lack of simple design abstraction that limits the adoption of analogue and mixed signal solutions [108]. Development of novel design pipelines, alongside new field programmable analogue arrays may address this issue, however the production of reliable and reusable parametrised analogue circuit blocks remains a considerable research challenge at this present time.

The dependence on fabrication process makes replication of published findings challenging and often leads to articles outlining the underlying theory rather than

detailing the implementation itself. In the case of the DPI circuit this may be seen in Bartolozzi and Indiveri's paper where the circuit diagram and underlying mathematical model are provided, without any attempt to detail the physical transistor scales used when developing the fabricated circuit [106]. This abstraction is necessary to ensure that the work remains independent of both process and fabrication scale, however it means that the application of analogue IC research findings will often involve considerable investments in both time and resource.

While it is disappointing that this test IC did not yield any experimental data or results, the issues identified with this work have helped inform research decisions. The importance of running LVS and DRC validation on all levels of hierarchy was identified, ensuring that this type of short circuit failure will not happen in future designs. Simulations across a wider process parameter variation must also be used to guarantee that nominal operation is achieved over the full expected process variation range. The challenges surrounding analogue abstraction and design replication were also demonstrated. Novel abstraction techniques and new field programmable analogue arrays are required to develop flexible and reproducible analogue neural networks, and considerable design challenges must first be overcome to meet these requirements [108].

The impact of process variation on analogue designs also represents a significant issue for large scale neural network implementation. As the number of neurons implemented on a single IC is increased, the variation in process parameters will lead to different neuron model responses. Some means of process parameter correction must be built into the system to ensure the network operates in a deterministic manner, allowing the individual neurons to be tuned or matched to one-another. Without this neuron tuning, each IC would have to be trained independently to account for differences in their internal operation.

5.2 Analogue CPG Design

A second analogue design was produced that formed part of the EPSRC CResPace project, aimed at developing adaptive bio-electronics for chronic cardiorespiratory disease. This project intends to produce an adaptive pacemaker, using CPGs formed of small neural networks to accurately reproduce biological motor sequences and mirror their adaptation to multiple physiological inputs. The IC was developed to support four analogue neurons, each broken out to the I/O of the chip, allowing the model parameters and resulting model performance to be explored. Section 5.2.1 discusses the model used to form the neurons, with Section 5.2.2 briefly reviewing the IC design and validation.

When developing medical devices it is important to consider how the proposed solution will impact pre-existing treatments and medical procedures. MRI techniques have been

developed by a number of groups to measure cardiovascular function during breathing cycles [109] and patients with cardiac and respiratory issues may therefore undergo MRI procedures during the course of diagnosis and monitoring. It is therefore important to explore how the magnetic fields generated by MRI scanners impact the operation of any proposed analogue CPGs. Section 5.2.3 details a set of tests investigating these effects, with experiments performed in a 3T static field generated using an MRI magnet. Experiments involving dynamic (imaging) magnetic fields require a qualified radiographer and access to a fully functional MRI machine. This greatly increases the experimental cost and delay and it is therefore sensible to first check the system in such static fields before moving to dynamic field tests.

5.2.1 Neuron Model

There are many different analogue neuron models and designs available [14]. As with the wide range of mathematical and computational neural models, these often try to emulate a specific element of the biological neurons function and must therefore be selected according to the design requirements. In the CResPace project, it is critical that the model generates and responds to biologically shaped APs as interfacing with the human body is the intended end application. This specification also means that the voltages, currents and timescales used within the system must be comparable to those seen within biological neurons.

For these reasons, a model proposed by Mahowald and Douglas (termed the MD-Neuron hereafter) was selected for its accurate emulation of the functional characteristics of real nerve cells [110]. As a sub-threshold analogue model, this design leverages the fact that the voltage-gated ionic channels of biological membranes in steady state display a sigmoidal conductance-voltage relationship. This may be shown to closely match that of a CMOS differential pair, yielding an efficient and direct silicon model [111].

The MD-Neuron circuit may be divided into distinct parts according to the implemented ionic currents. This means that designers may add additional ionic currents as desired, with minimal changes to the existing model framework. As with many other neural models, the original MD-Neuron supports both potassium and sodium conductance. In order to provide the relatively slow ionic current variables, common to neural models, the MD-Neuron implements a set of low-pass filters in the form of follower-integrator circuits, shown in Figure 5.4. These filters use a bias current, I_{bias}, to control the time constants of the variables in question.

The Potassium Conductance

As described in Chapter 2, activation gates contribute to the conduction of ions, while inactivation gates close the channels. The potassium channels use only activation gates

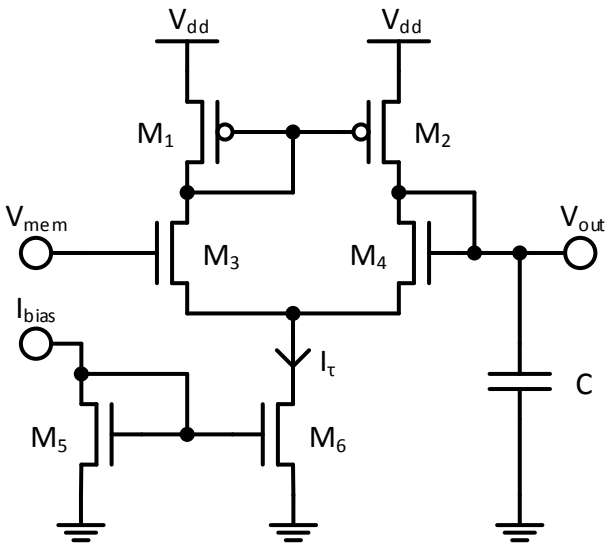


Figure 5.4: The follow-integrator circuit, used as a low-pass filter to control the slower ionic variables in the MD-Neuron. I_{bias} controls the circuits time constant, τ, while V_{mem} represents the membrane potential and V_{out} yields the slower filtered membrane potential.

and are therefore somewhat easier to implement. Figure 5.5 shows the potassium channel circuit, with the underlying sub-circuits identified by boxed off regions. This circuit uses the membrane potential, V_{mem}, to control the potassium current sink, g_k. When the conductance circuit (region C) is activated, the current sink, g_k, draws current off of the membrane capacitor (shown in Figure 5.7). This results in repolarisation of the membrane potential.

The potassium conductance circuit is controlled using a differential pair (region B) connected via a current mirror (region A). The differential pair receives a low-pass filtered version of the membrane potential, V_{mem}, slowing the circuits response to fluctuations in the membrane potential. This filtered input is compared against a reference voltage, V_{knee}, while the bias current, I_{K_max}, defines the sum of the currents flowing through the differential pair transistors (M_3 and M_4).

The Sodium Conductance

The sodium channel circuit, shown in Figure 5.6, must include both activation and inactivation gates. In the case of sodium channels, the activation gates response is almost immediate whereas the inactivation gate response is delayed. To account for this time difference, the inactivation circuit makes use of another low-pass filter to delay the membrane potentials effect on sodium inactivation. No filter is required for the activation circuit since the activation response of sodium channels follows the membrane potential directly.

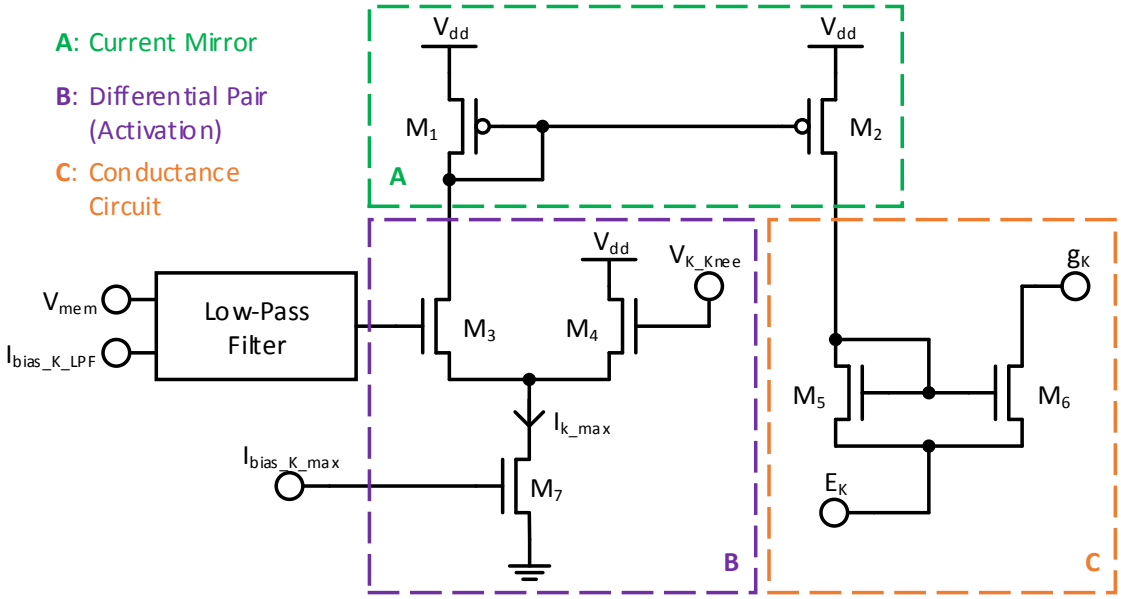


Figure 5.5: The MD-Neuron's potassium channel circuit where a differential pair is used to model the activation gates response to a delayed membrane potential. The desired time-constant for the low pass filter is set using $I_{bias\_K\_LPF}$. $I_{bias\_K\_max}$ and V_{K_knee} determine the activation gates response to stimulus and E_K represents the potassium Nernst potential.

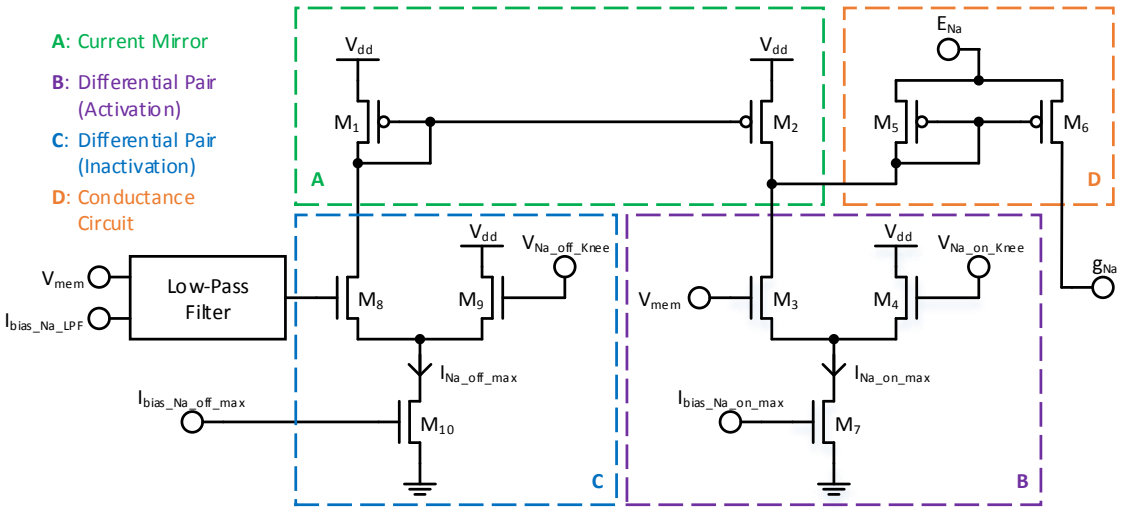


Figure 5.6: The MD-Neuron's sodium channel circuit where a set of differential pairs are used to model the activation and inactivation gates response to a delayed membrane potential. The desired time-constant for the low pass filter is set using $I_{bias\_Na\_LPF}$. $I_{bias\_Na\_off\_max}$ and $V_{Na\_off\_knee}$ determine the inactivation gates response to stimulus; $I_{bias\_Na\_on\_max}$ and $V_{Na\_on\_knee}$ determine the activation gates response to stimulus; and E_Na represents the sodium Nernst potential.

The current mirror and activation differential pair used in this circuit are functionally identical to those of the potassium channel circuit. The conductance circuit operates in a similar fashion to before, however in this case the sodium conductance, g_{Na}, acts as a current source and drives voltage onto the membrane capacitors when activated.

To implement the overriding effects of inactivation, a second differential pair is added to this circuit. When the delayed membrane potential exceeds the inactivation reference potential, $V_{Na\_off\_knee}$, current begins to flow through M_1. This causes more of the current $I_{Na\_on\_max}$ to be sourced through transistor M_2, reducing whatever current was previously flowing through M_5. In this way, the inactivation circuit can stop sodium conductance regardless of the channels activation state.

Additional Ionic Currents

Additional ionic currents may be added to this model by using Figures 5.5 and 5.6 as templates. In this way, channels operating as both current sources and current sinks may be added with, or without, inactivation gate support.

The Membrane Potential

As suggested during the description of the ionic channels, the membrane is modelled using a capacitor. Each of the ionic channels then either inject or drain current on this capacitor, modifying the membrane potential V_{mem}. This setup is shown in Figure 5.7 and represents the complete MD-Neuron model.

5.2.2 MD-Neuron IC Design and Validation

With the model identified, a $0.35\mu m$ design for an IC was produced using the circuits described in Section 5.2.1. Each of the subsystems were tested using ideal current and voltage sources to ensure that the transistor dimensions were correctly sized. These subsystems were then combined into a final design, replacing the current sources with current mirrors and external connections where required. This IC was designed to contain four neurons in total. Two of these neurons were modified to operate as voltage clamped neurons, meaning that the voltage of the membrane may be set and the associated ionic currents monitored. The other two neurons operate as standard neural models, as intended for the final CResPace project. These four neurons were arranged into a two-by-two grid taking up the majority of the available floorspace.

The default values used for each bias current and reference voltage are provided in Table 5.1. Current divider circuits were added to the design, allowing input currents to be reduced by a factor of 10 or 100, ensuring that the smaller bias currents of $1\mu A$

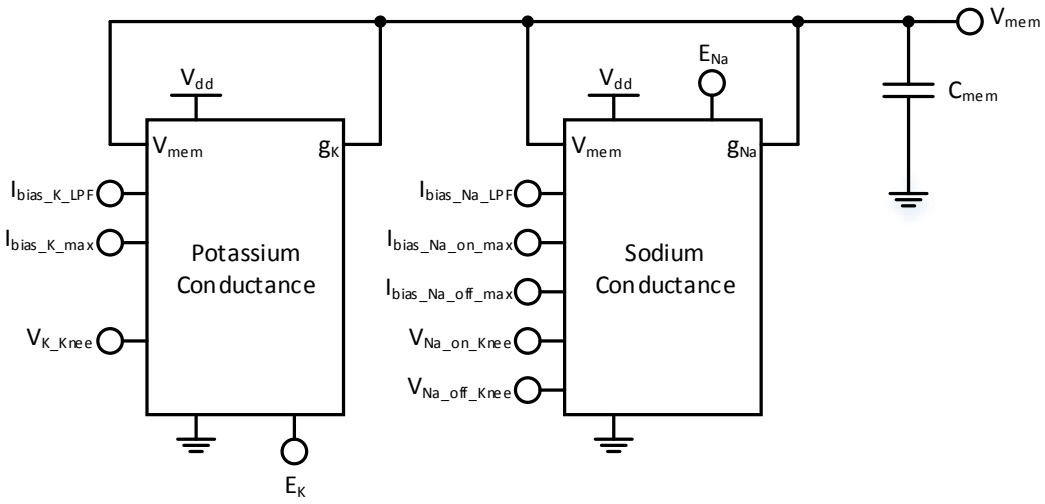


Figure 5.7: Top-level layout for the MD-Neuron, showing how the independent ionic channel circuits are connected via the membrane potential, V_{mem}. A membrane capacitor, C_{mem}, converts the ionic currents into membrane potential fluctuations. Currents may be injected onto the membrane capacitor to emulate external stimulus.

Table 5.1: Bias currents and voltages used in the implementation of the MD-Neuron. The smaller current values of $1\mu A$ and $100nA$ were produced using sets of current dividers driven using $10\mu A$ current sources. Voltages are referenced to analogue ground.

Name	Value	Name	Value
$I_{bias\_K\_LPF}$	$100nA$	$I_{bias\_Na\_LPF}$	$1\mu A$
$I_{bias\_K\_max}$	$10\mu A$	$I_{bias\_Na\_off\_max}$	$10\mu A$
-	-	$I_{bias\_Na\_on\_max}$	$10\mu A$
V_{K_knee}	$0V$	$V_{Na\_off\_knee}$	$0V$
-	-	$V_{Na\_on\_knee}$	$0V$
E_K	$-0.5V$	E_{Na}	$0.5V$

and $100nA$ could be reliably provided using external stimulus. As a result, the bias currents required at the chip's I/O level are all set at $10\mu A$.

Spectre Design Validation

The completed design was simulated using Cadence Virtuoso Spectre models to ensure that the neuron operated according to the design specifications. To perform these tests a set of ideal voltage and current sources were connected to the neurons I/O, as shown in Figure 5.8. A current pulse source was used as the input for the system and two parameters, T_{width} and I_{max}, were used to control the shape and scale of the excitatory inputs.

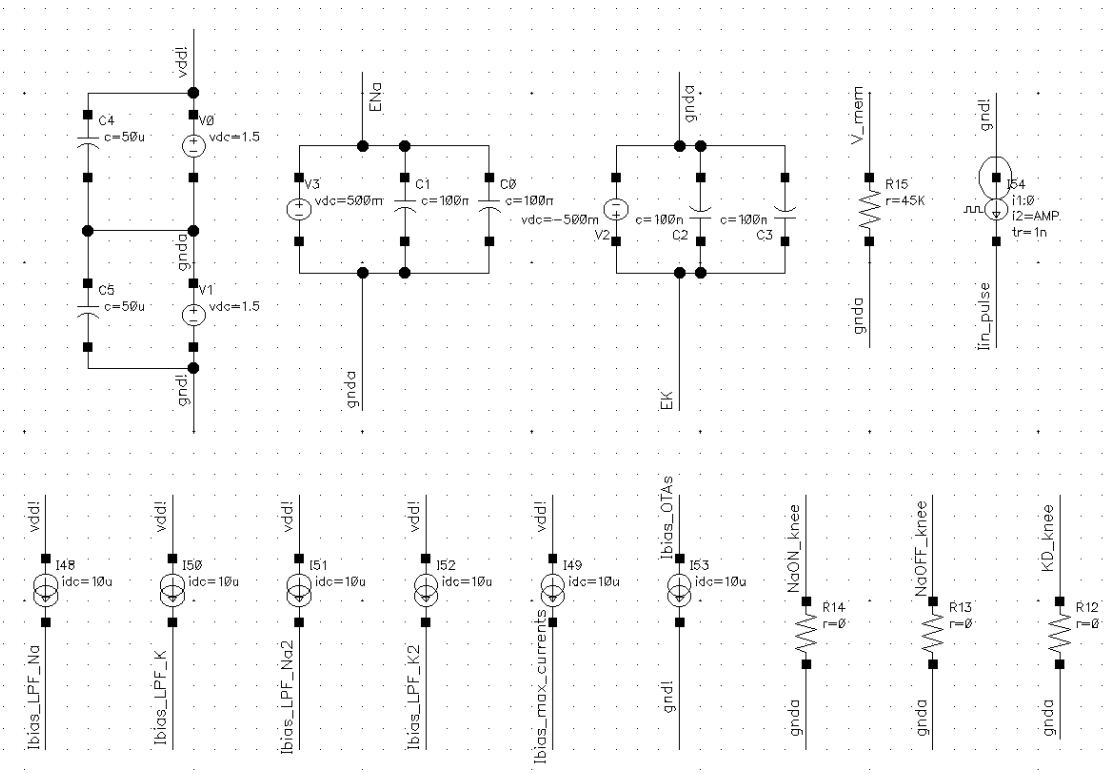


Figure 5.8: Neuron test bench setup with ideal current and voltage sources used to set the various bias values. Decoupling capacitors were included on the voltage rails at this stage to ensure that their later addition to the MD-Neuron IC wouldn't negatively interfere with the designs overall function.

Three different modes of operation were tested using this setup. First, the neurons response to sub-threshold stimulus was verified, using a $100\mu s$, $5\mu A$ pulse. For this test the neuron showed slight depolarisation before restoring to the resting potential, as shown in Figure 5.9a. Next, a supra-threshold $100\mu s$, $10\mu A$ stimulus was applied. In this test the neuron generated a full AP. Shown in Figure 5.9b, the generated AP continues beyond the time-frame of the input stimulus, as observed in biological neurons. Finally, the neurons response to sustained stimulus was considered via the application of a supra-threshold $10ms$, $10\mu A$ pulse. In this test, the neurons produced a tonic spiking signal shown in Figure 5.9c. This spiking pattern persisted throughout the stimulus period before returning to a resting state when the input ceased.

Chip Layout

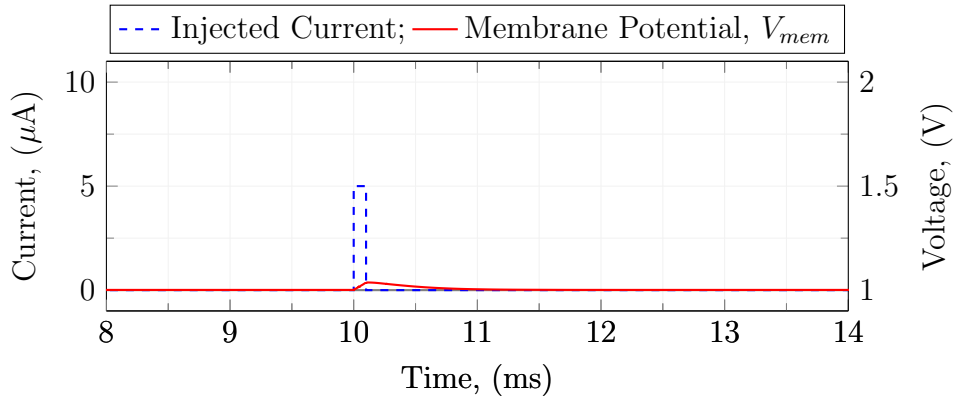
With the MD-Neuron IC design verified, the layout process was performed. At the request of project collaborators, the membrane capacitors (C_{mem} in Figure 5.7) were added on-chip. These capacitors are much larger than the transistors and therefore represented a significant portion of the required floorspace for each neuron. The final layout is shown in Figure 5.10.

Once the neuron layout was finalised, the chip layout was generated by arranging multiple instances of these neurons around the available floorspace. Shown in Figure 5.11, the final chip contained 4 individual neurons. Two of these neurons were modified to operate as voltage clamped neurons in an effort to support wider experimentation when using the chips. Voltage clamped neurons have their membrane potential locked at a target voltage allowing the resulting membrane currents to be measured. This measurement technique allows the relationship between membrane potential and ionic currents to be directly recorded.

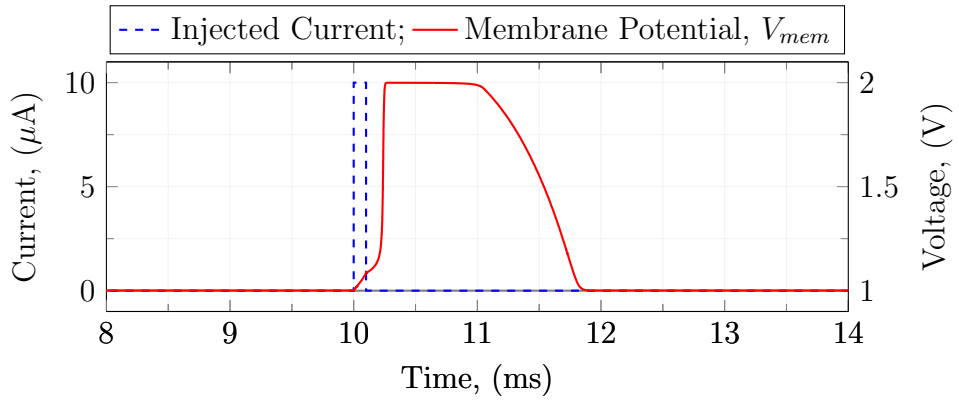
As previously mentioned, the capacitors represent a significant portion of the design floorspace. This significantly limits the number of neurons that may be arranged into any given die and demonstrates another common issue with analogue IC design. In the case of CPG emulation, which typically only contain a small number of neurons, this scaling limitation is not a constraining factor and this design is therefore suitable for its intended purpose.

Simulating the Test Setup

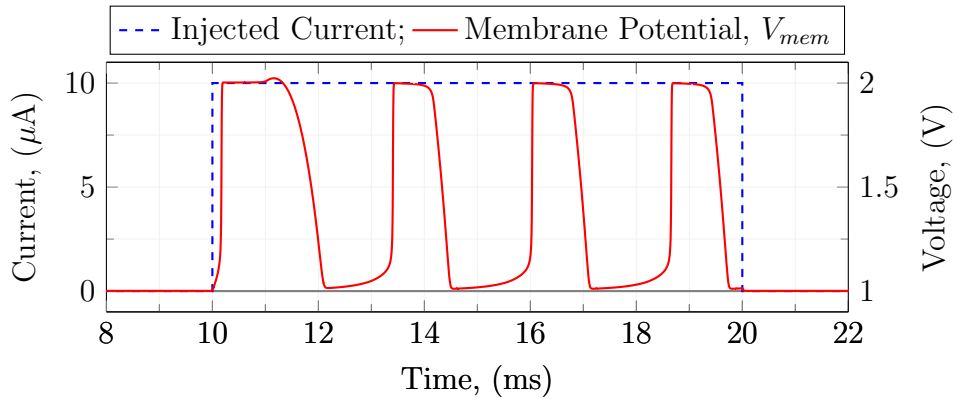
Plans to test the chips were drawn up while the IC underwent fabrication. This first required a modification of the original test setup shown in Figure 5.8 to replace the ideal sources with better representations of the test equipment itself. As shown in Figure 5.12, current sources were replaced with resistive loads pulled to the appropriate voltage rail, the Nernst potentials were generated using potential dividers and the input supply was achieved through the application of a signal generator.



(a) MD-Neuron's response to sub-threshold stimulus, showing slight depolarisation before undergoing full restorative repolarization.



(b) MD-Neuron's response to supra-threshold stimulus, showing full generation of an AP.



(c) MD-Neuron's response to an extended supra-threshold stimulus, showing tonic spiking patterns that persist throughout the entire stimulus period.

Figure 5.9: MD-Neuron's simulated response to different input stimuli, generated using Spectre in Cadence Virtuoso.

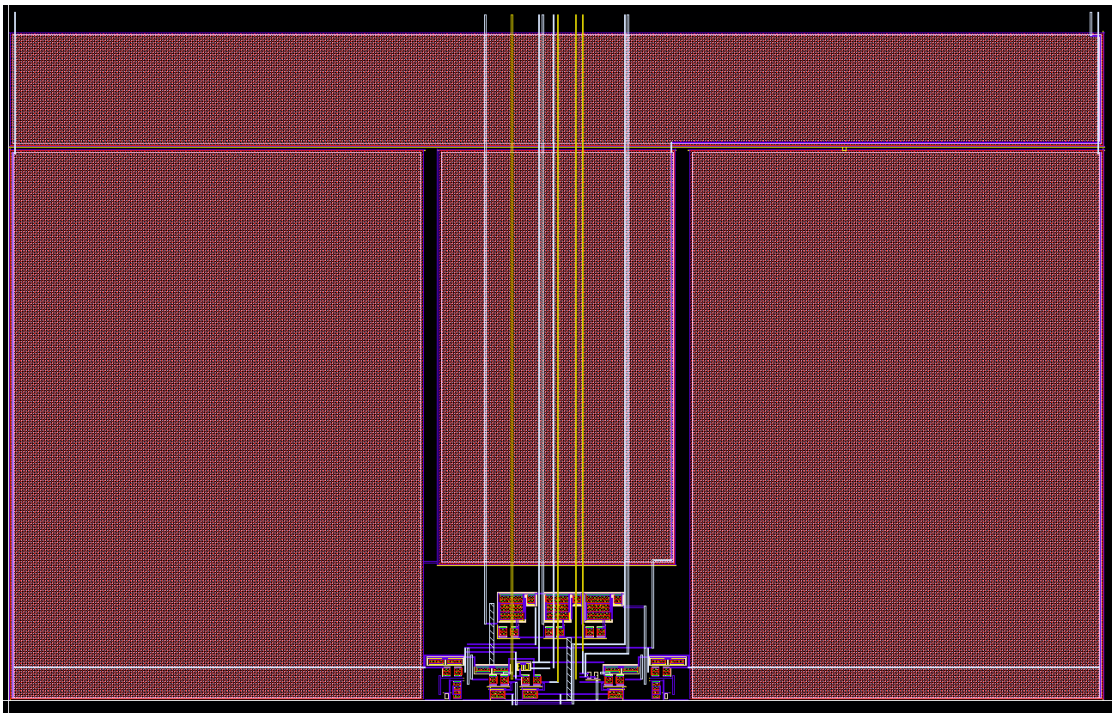


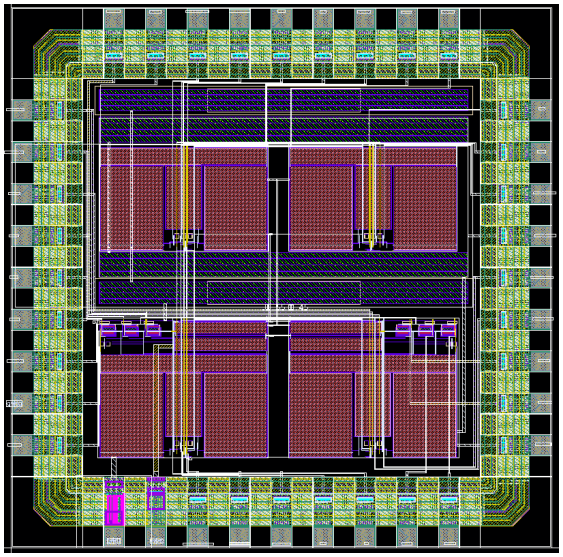
Figure 5.10: Layout for an individual MD-neuron, showing the relatively large capacitors required to enable full ASIC implementation of the design.

These modifications to the test setup are critical in ensuring that the simulated results are meaningful when compared against the experimental results.

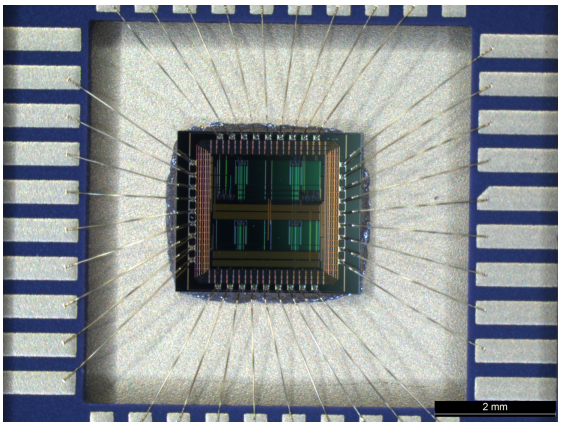
Monte-Carlo Simulation

When fabricating ICs, process variation plays a key role in determining the success and failure rates of a design. Sub-threshold designs, such as the MD-Neuron IC, are particularly susceptible to process variation as the transistors in such designs are operating in their non-linear region. A small change in the transistors parameters can therefore have a dramatic impact on the transistors output for a given input value. This means that small process variations can entirely change the dynamics of the system as a whole. Designs must therefore be tested to ensure that the expected level of variation does not result in failures. Biological neural systems, however, demonstrate a high noise tolerance and reliable performance in changing conditions. Replication of this robustness is a key motivator in neuromorphic hardware design, with the hope that future systems will be capable of active adaption to less-than-ideal initial conditions.

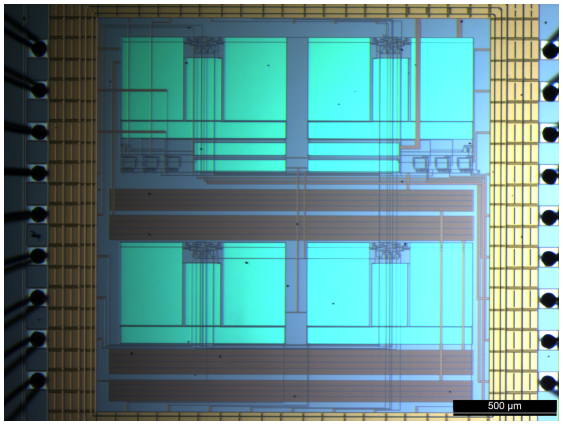
Monte-Carlo simulations are commonly used to assess the impact of process variation. These tests perform a large number of simulations, tweaking the selected parameters to explore the problem space. From these simulations, designers may assess how process



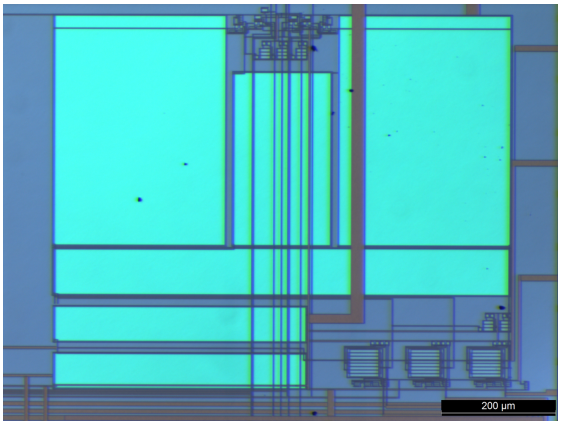
(a) Final Layout



(b) Bonded Chip



(c) Fabricated Die



(d) Fabricated Neuron

Figure 5.11: Final MD-Neuron IC layout, showing four individual neurons arranged in a 2×2 grid. (a) shows the layout as seen in Cadence; (b) shows a photo of the bonded die; (c) shows the die itself, as viewed through a microscope; and (d) shows an individual neuron.

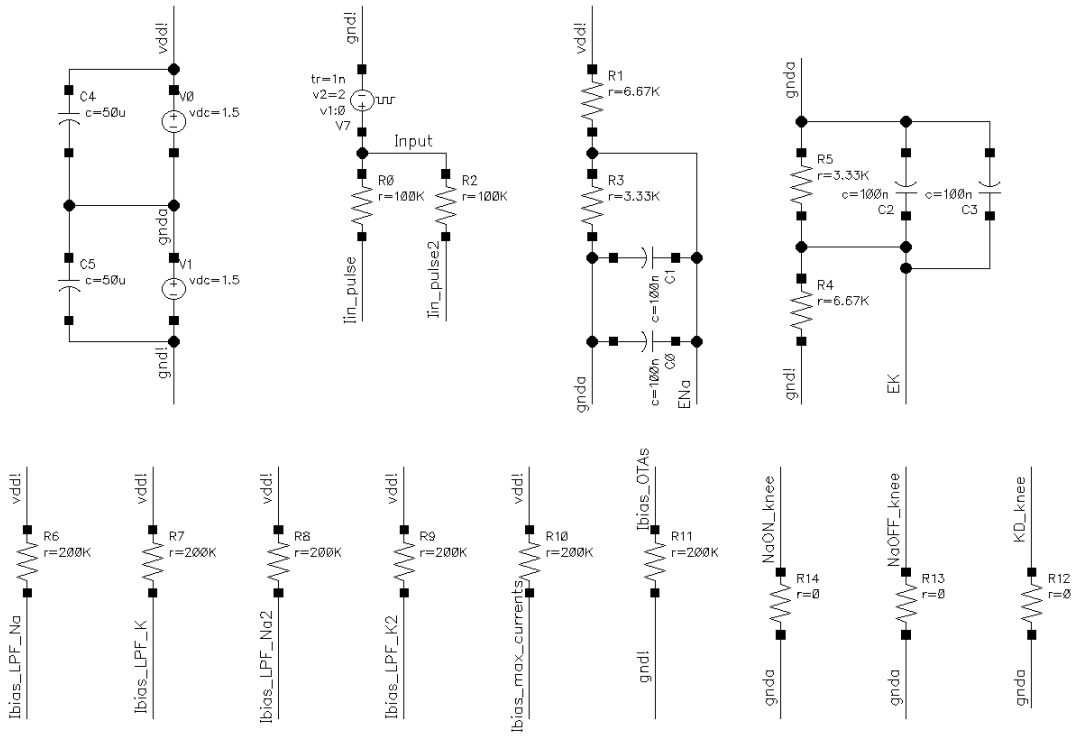


Figure 5.12: The experimental setup was reflected by changes to the simulated test bench. This helped reduce the risks of variation between experimental and simulated results by removing the ideal current and voltage sources.

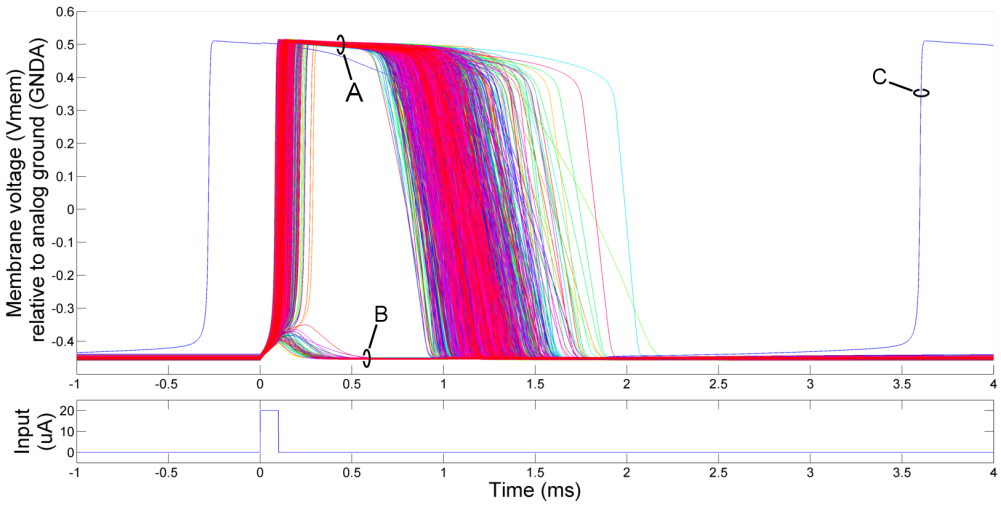


Figure 5.13: Monte-Carlo simulation of MD-neuron APs for a 100μs, 20μA pulse stimulus.

variation will affect their designs.

A Monte-Carlo simulation was performed using the experimental setup shown in Figure 5.12. This Monte-Carlo simulation used 750 points, an amount chosen to ensure that the simulation could be performed within the available compute time on the Cadence server. Since a single experimental rig would be used to test the ICs, the bias currents and chemical potentials may be assumed constant across all ICs and therefore remain constant during the entire Monte-Carlo simulation. As this test was performed at chip-level, there are two neurons to each Monte-Carlo simulated point.

To ensure that an AP would be produced by the majority of neural circuits under test, a 100μs supra-threshold input pulse of 20μA was simulated. Figure 5.13 shows the membrane potentials for all 1500 neural circuits following this 20μA, 100μs pulse. This graph shows three distinct classes of operation. The first (marked as group A) encapsulates all neurons which have fired in response to the stimulus. This group contains 1450 of the simulated neurons. The second class (marked as group B) contains all neurons which have undergone a small depolarization when stimulated, but rapidly return to the resting potential. The final class of operation contains a single neuron (marked C), which may be seen to undergo periodic AP generation regardless of input stimulus. Such neurons may be identified rapidly since they fire even when no stimulus is applied.

Focusing on neurons which fall into class A, it may be noted that the falling edge of the AP is most susceptible to timing drift caused by process variation, while the upstrokes are relatively similar. The resting potential and maximum firing potential are set by the sodium and potassium potentials and are therefore largely unaffected by process variation, however it may be noted that the resting potentials appear to pinch together following an AP. This is likely due to the limit caused by the sodium potential, which

stops the neurons from entering hyperpolarization.

These results show that there is significant variation in AP morphology caused by process variation, with a total of 3.3% of the neurons failing to fire at all, and 1 neuron entering a complete failure state of periodic AP generation. This means that these neurons will require careful tuning to ensure that they generate the desired APs, however complete failure of the model is unlikely, with a predicted successful yield of 99.4%.

Experimental Setup

With the experimental setup simulated, an automated test bench was assembled to allow the rapid testing of 20 MD-Neuron ICs, each containing two complete neural circuits. As before in Figure 5.12, the supply rails were set to $\pm$1.5V using a bench power supply, with decoupling capacitors to reduce line distortion. The sodium and potassium potentials were set to a fixed $-$500mV and 500mV respectively using potential dividers and decoupling capacitors as simulated. Finally the inputs were connected to a signal generator, using resistors to provide a simple current source. Each of the bias currents were set to 10μA in order to match the previously simulated system.

An oscilloscope was used to record the neural signals produced in response to a given stimulus. This oscilloscope and the input signal generator were connected to a computer allowing automated stimulus generation and response recording.

The test system was programmed to produce a sequence of input stimuli, using 100μs pulses ranging from 2μA to 20μA in steps of 0.25μA. The ICs response to each stimulus was recorded for later analysis. Once all stimuli tests were performed the IC was swapped and the process repeated until results for all 20 neural ICs had been collected.

Experimental Results

Following collection, the experimental data was processed to detect any neurons operating incorrectly. Of the 40 neurons tested, 2 were found to be operating within class C outlined in Section 5.2.2. The other 38 neurons were all found to operate within class A for a stimulus pulse of 20μA, as shown in Figure 5.14.

As discussed in Section 5.2.2, it may be noted that the falling edge of the AP is most varied from IC to IC while the rising edge or upstroke is more consistent between ICs. The resting potential and maximum firing potentials are again consistent between ICs, with the same pinching of resting potential visible when you compare the resting potential variance before and after an AP has occurred.

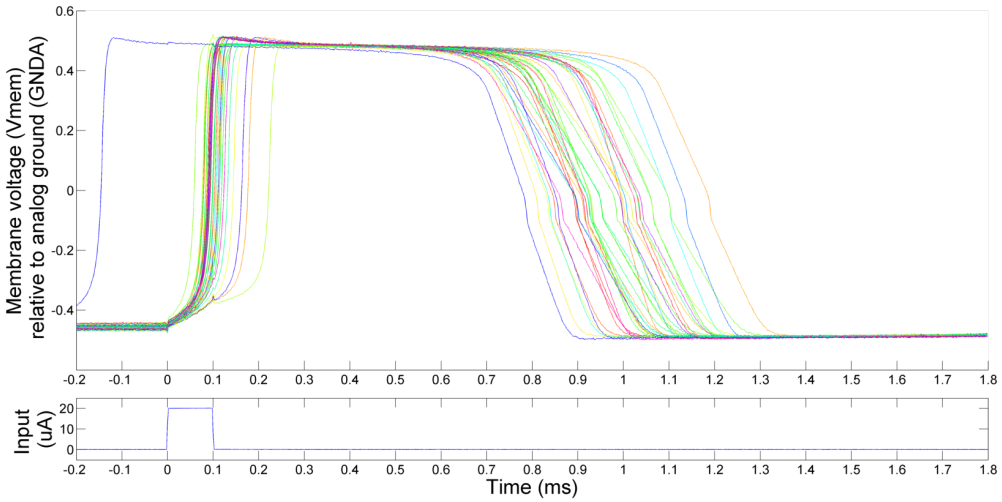


Figure 5.14: MD-neuron IC APs for a 100μs, 20μA pulse stimulus.

Results Analysis

Both the simulated and experimental results showed class A and C responses to a 20μA stimulus. While class B responses were not seen in the experimental data at 20μA, lower stimuli values were seen to yield a class B response suggesting that the simulated class B neurons have simply had their threshold level shifted by process variation. From Figure 5.13 it is apparent that class B responses form a small sub-set of the 1500 neurons under test and it is therefore unsurprising that the 40 physical neurons have not demonstrated such significant threshold level changes.

The results may be better compared by plotting the AP averages and AP timing windows as shown in Figures 5.15 and 5.16. In these plots it is clear that the experimental data easily falls within the domain of the simulated data. This is to be expected due to the difference in sample sizes between the 1500 simulated neurons and the 40 physical neurons. The falling edges of both the simulated and experimental APs were prone to timing drifts caused by process variation, with the bulk of the simulated results and all the experimental results falling within a 500μs window. The rising edge of the AP varied by about 250μs for both the simulated and experimental results. A close inspection of the falling edges in Figures 5.13 and 5.14 also reveals that the falling gradient is also prone to some variance, however this gradient seems more consistent than the timing across the full range of Monte-Carlo results.

The voltage levels are set by the Nernst potentials and show high process variation tolerance in both the simulated and experimental data. The modification to the width of the APs is likely due to fluctuations in the time constant of the low-pass filters used to generate the delayed membrane signals. A process variation tolerant integrate-follow filter must therefore be designed if greater consistence between models is required.

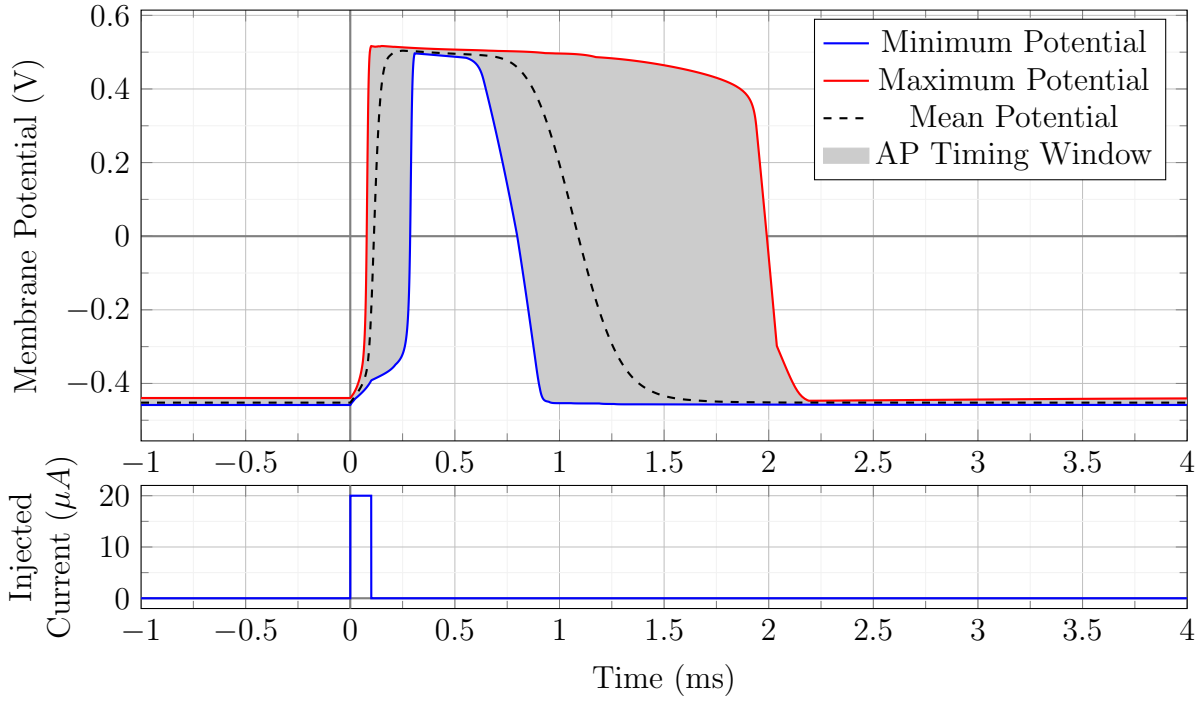


Figure 5.15: MD-neuron AP timing window according to a Monte-Carlo simulation of 1500 APs generated in response to a $100\mu s$, $20\mu A$ pulse stimulus.

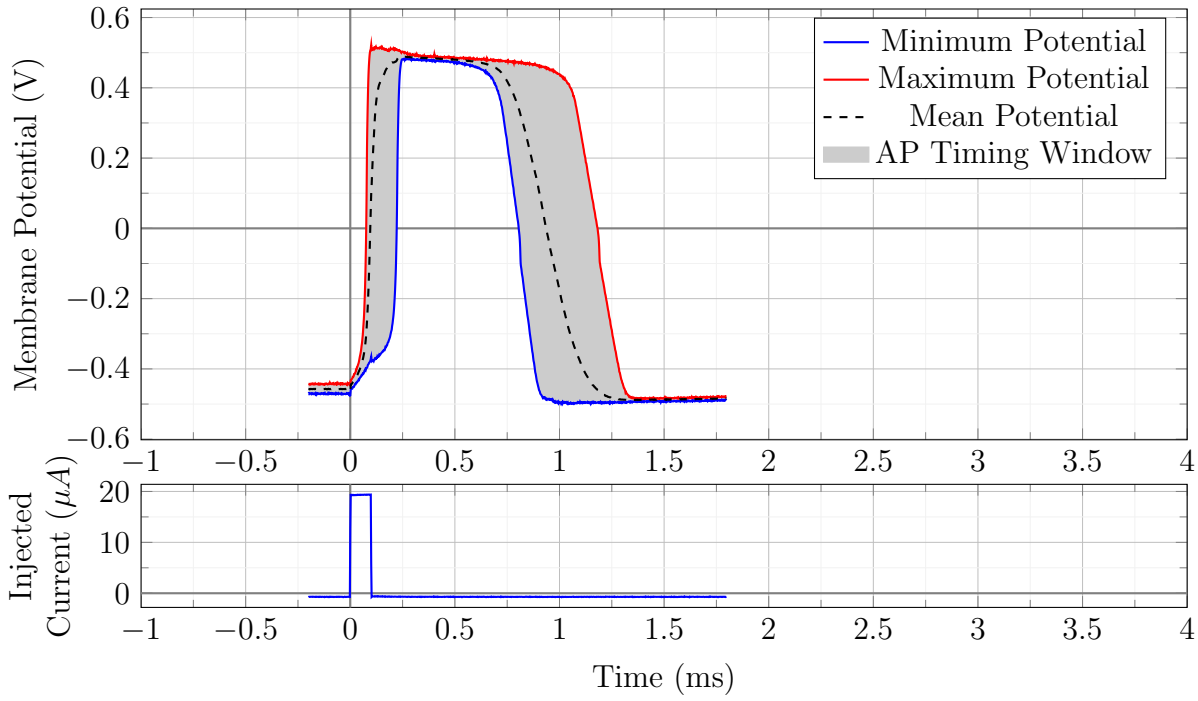


Figure 5.16: Experimental MD-neuron IC AP timing window according to the measurement of 40 APs generated in response to a $100\mu s$, $20\mu A$ pulse stimulus.

A small zero-crossing error may also be seen in the experimental results. This perturbation was not seen in the simulated results and it was later discovered that grounding issues in the signal generator and power supplies was the principle cause of this divergence. Since the edge remains continuous it may be shown that any gradient following or zero-crossing detectors would still function correctly in the presence of this perturbation.

The simulated results closely match that of the experimental results suggesting that the Monte-Carlo simulation provides a good representation of the expected performance for this model of neurons. Two of the forty IC neurons, however, were found to be in a failure state before connecting any input to the system, while only 1 of the 1500 neurons in the simulated tests was found to be in a state of failure. This suggest that either the Monte-Carlo simulation has failed to identify a cause of neuron failure or the chip batch itself was sitting close to the boundary of failure. It must also be considered that the physical fabricated chips may fail due to other contributing factors outside of process variation itself. For these reasons, the relatively high failure rate is not overly concerning when compared against the simulated results.

Many of the changes in model performance may be undone through careful calibration of the bias currents and voltages. In this way, one neuron may be made to operate in a functionally similar fashion to that of another. This MD-Neuron IC may therefore be used to generate hand-tuned CPGs for application in experimental medical devices.

5.2.3 Operation Within High Magnetic Fields

It is important to consider how new medical devices may be affected by existing treatments during the development stages. One of the CResPace projects key goals is to develop a pacemaker solution that operates nominally within MRI scanners. This task poses a significant technical challenge due to the large static and dynamic magnetic fields used for MRI procedures. Unaccounted for, these fields can cause critical failure in electronics or physical damage if the device contains sufficient ferrous material. If successful, the CResPace project will revolutionise cardiorespiratory care, providing a practical way for doctors to monitor patients who have active medical implants such as pacemakers.

MRI scanners use a high static magnetic field of between 1T and 7T to align the protons in water molecules within the body, with most clinical systems at either 1.5T or 3T. A dynamic field of around 30mT is then used to agitate this alignment. The time taken for the atoms to re-align to the static field provides information on the structure and contents of the scanned area.

The high static fields make MRIs procedures unsuitable for any individual with surgical implants that contain magnetic elements. Additionally, the dynamic fields can induce

eddy currents in conductive elements which will both interfere with electronic systems and heat the component. The strength of the static field means that large conductive parts can produce considerable torque or resistance to motion when moved through the field lines. These factors make MRI procedures unsafe for individuals who have medical implants.

When designing a system that can be used safely within an MRI scanner there are two core challenges. Firstly, the system must be safe for insertion into high magnetic fields. This requires non-magnetic components and small metallic surface areas. Secondly, the system must be unaffected when subjected to both static and dynamic magnetic fields.

Method

The following section describes an experiment that was developed to test the MD-Neuron chip design in high magnetic fields. An NMOS test structure was first profiled within the field to provide insight into the underlying mechanics of any operational change. This test structure contained a single 10/2 NMOS transistor on the $0.35\mu m$ technology. An electromagnet capable of generating a 3 T static magnetic field was provided by Siemens MR Magnet Technology, shown in Figure 5.17. This coil generates a spherical region of uniform field about 1m in diameter.

This NMOS test was performed before the MD-Neuron IC test to measure the impact of the magnetic field on a single transistor. It was possible that the hall effect could cause changes in the transistors operation, leading to different transconductance properties. These changes could cause higher-level circuits to fail and it was therefore necessary to characterise any change in the transistors to enable further changes in the MD-Neurons operation to be diagnosed and modelled. In the event of complete neuron failure, this NMOS data could be used to build a Spectre or PSPICE model, allowing future magnet-resistant designs to be developed and verified.

The NMOS test structure was profiled using a Source Measurement Unit (SMU) and 3V DC supply setup as shown in Figure 5.18. Programs were developed for the SMU to automate the output and transfer characteristic recordings. The transfer characteristics are taken for the range $V_{ds} = 0 \text{ to } 3v$ with V_{gs} values of $[0.00, 0.25, 0.50, 0.75, 1.00, 1.25, 1.50]$. This process is repeated 10 times to ensure that the impact of thermal effects and external noise is mitigated.

The output characteristics are profiled at $V_{ds} = 3v$, sweeping V_{gs} from 0V to 3V in increments of 0.01V. In both experiments the test chip must be powered for a couple of minutes prior to the readings to ensure that the system had reached a thermal equilibrium.

Each experiment is performed a total of four times, once outside the influence of the

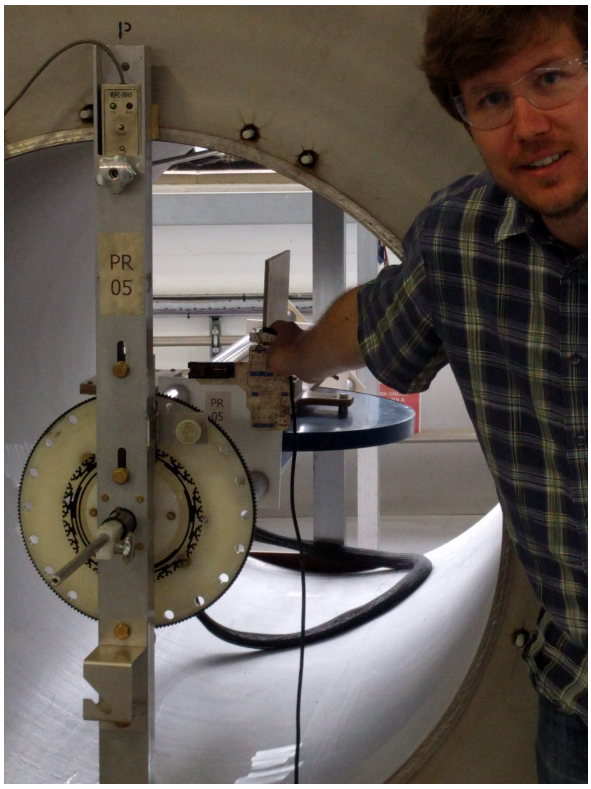


Figure 5.17: The test circuit board held within the electromagnetic coil capable of generating a 3T static magnetic field, provided by Siemens MR Magnet Technology.

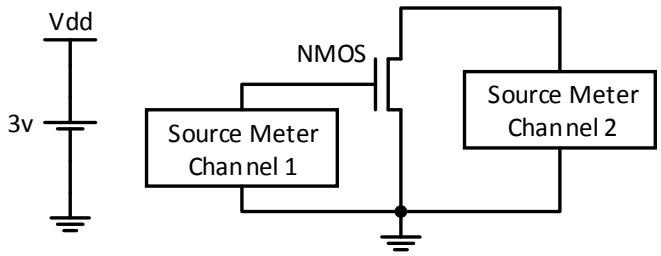


Figure 5.18: Electrical setup for the NMOS test structure experiments. In these experiments the output and transfer characteristics were recorded while under the influence of different magnetic field orientations.

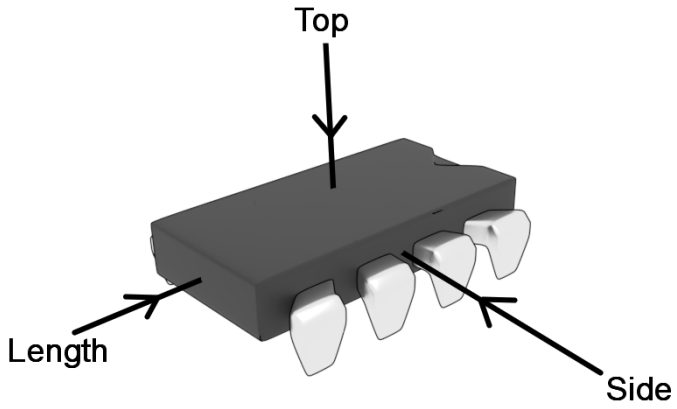


Figure 5.19: The three orientations for the magnetic field lines, shown as black arrows with their associated labels.

magnetic field and at three orientations within the field. These orientations are notated as 'Top', 'Length' and 'Side', shown in Figure 5.19 where the magnetic field lines are notated by the arrows in each case.

The MD-Neuron chip was also tested using the three field orientations. For the first MD-Neuron test, a pulse stimulus is generated using an SMU. This pulse stimulus drives the two neurons incorporated within the MD-Neuron IC. An oscilloscope provides a recording of the neurons membrane potential, V_{mem}, allowing the neurons stimulus response to be assessed. The SMU is programmed to generate $100\mu s$ pulses that range from $10\mu A$ to $50\mu A$. This setup tests the neurons AP generation in response to sub- and supra-threshold stimuli.

For the second MD-Neuron test an extended stimuli set of $10ms$ width was generated. This setup tests the neurons tonic spiking generation in response to continuous sub- and supra-threshold stimuli.

In all three tests the out-of-magnet readings and three orientation readings were recorded on the same day to reduce the impact of external factors. Prior to any experiments, the magnet was slowly approached with both the test board and MD-Neuron IC to ensure that the mechanical forces were within a safe threshold. The IC was passed through the strongest region of magnetic field to detect if there was any significant motive force on the device. These safety checks confirmed that there was minimal ferrous material within the MD-Neuron IC making it safe for insertion within the MRI static coil. Following the static field tests described in this work, further experimentation will be required to assess the electrical and thermal effects caused by dynamic fields.

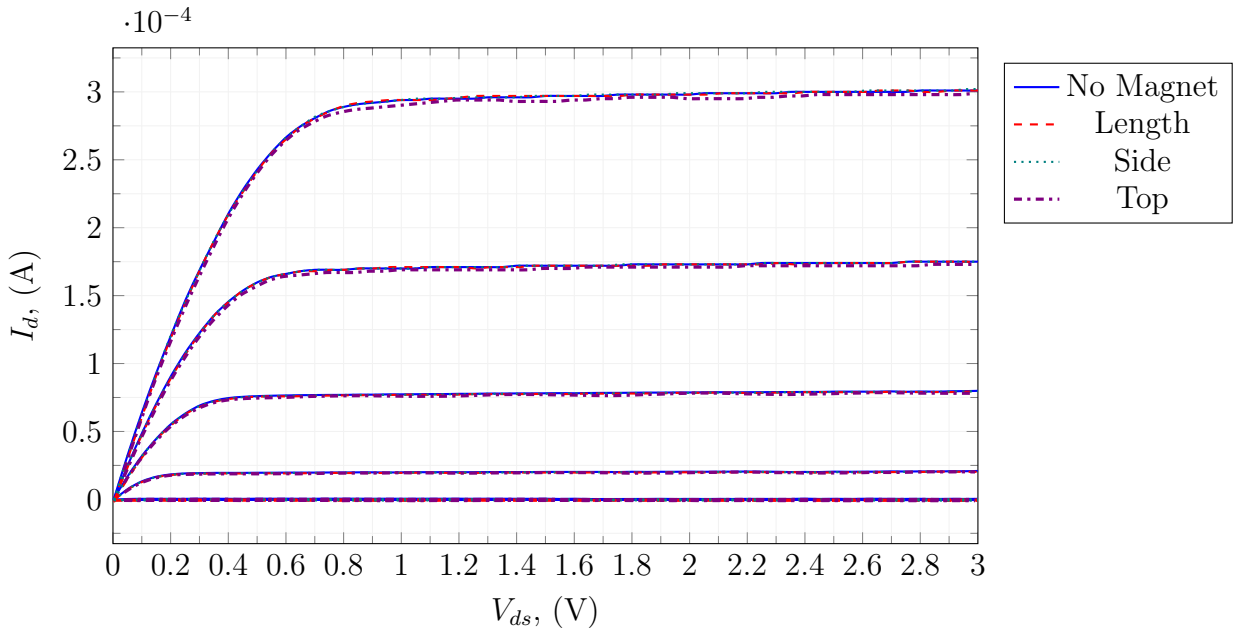


Figure 5.20: NMOS transfer characteristics in a static 3T magnetic field for V_{gs} values of [0.50, 0.75, 1.00, 1.25, 1.50] from bottom to top (Average value from 10 reading cycles).

NMOS Results

The NMOS tests are divided into two separate sets of experimental results. First the transfer characteristics are shown in Figure 5.20. From these results it appears that there is little-to-no change in the transistors transfer characteristics when situated in a large static magnetic field. Figure 5.21 shows a zoomed portion of the $V_{gs} = 1.50$v curve. From this figure it is seen that a chip with a Top oriented magnetic field sees a slight reduction in transconductance when compared against the other readings. This shift in I_d occurs in each of the traces but is most apparent for larger values of V_{gs}.

It is possible to calculate the experimental threshold voltage, V_T, using these V_{ds} traces in the first order approximation of the MOSFET equation shown in Equation 5.3.

$$I_d = \frac{1}{2} K_n (V_{gs} - V_T)^2 \left[1 + \lambda (V_{ds} - V_{dsSat})\right] \tag{5.3}$$

Selecting two traces for a given V_{ds} value, Equation 5.3 may be rearranged, with common terms cancelling out as follows:

$$\frac{I_{d1}}{I_{d2}} = \frac{(V_{gs1} - V_T)^2}{(V_{gs2} - V_T)^2} \tag{5.4}$$

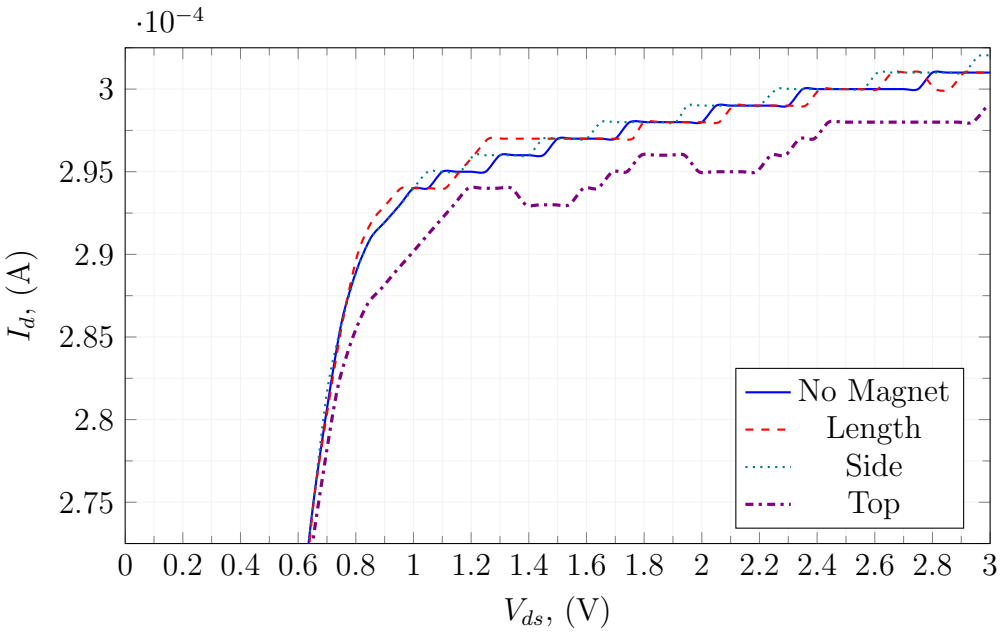


Figure 5.21: Zoomed NMOS transfer characteristics, where $V_{gs} = 1.50$v (Average value from 10 reading cycles).

Table 5.2: Experimental threshold voltage for a 0.35μm 10/2 NMOS transistor in differently oriented static 3T magnetic fields.

Orientation	V_t
No Magnet	0.570 ± 0.005
Top	0.572 ± 0.006
Side	0.571 ± 0.003
Length	0.570 ± 0.005

This Equation may be solved for V_T yielding the experimental threshold voltage.

$$V_T = \frac{\sqrt{I_{d1}I_{d2}(V_{gs1} - V_{gs2})^2} + I_{d1}V_{gs2} - I_{d2}V_{gs1}}{I_{d1} - I_{d2}} \tag{5.5}$$

The threshold voltage for the NMOS transistor in each field orientation is shown in Table 5.2. These values were calculated using a value of $V_{ds} = 3$v alongside the readings from two traces $V_{gs1} = 1.5$ and $V_{gs2} = 1.25$. The threshold voltage calculated from the experimental results closely matches that of a simulated $0.35\mu m$ 10/2 NMOS transistor, with an expected threshold voltage of $V_{t(sim)} = 0.565$v.

The output characteristics are shown in Figures 5.22 and 5.23. From these results there is no apparent change in the NMOS output characteristics when at field. These reading were taken with a V_{ds} value of 3v.

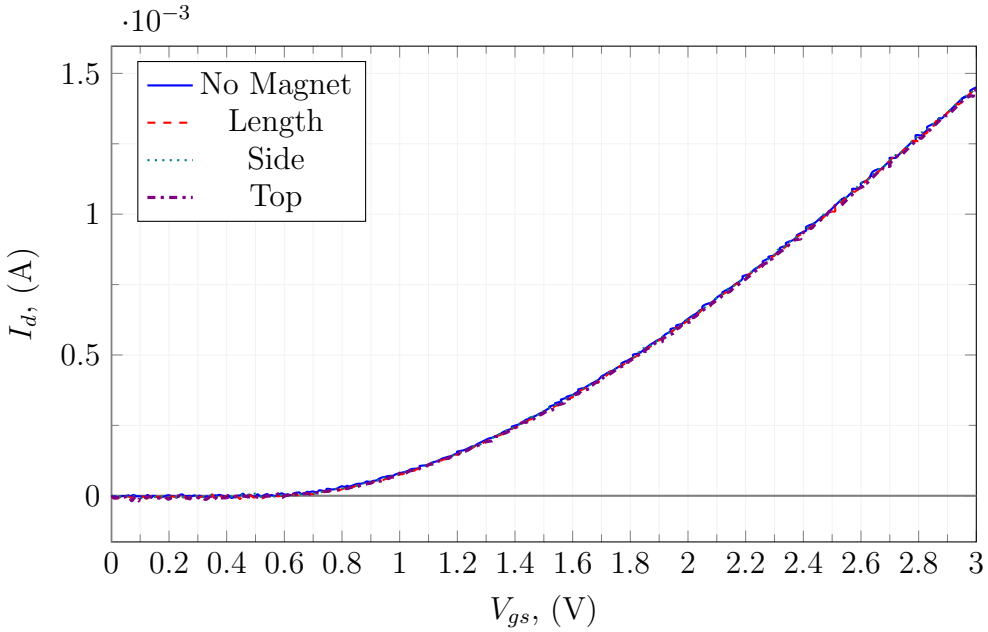


Figure 5.22: NMOS output characteristics show identical characteristics when both out- and in-field.

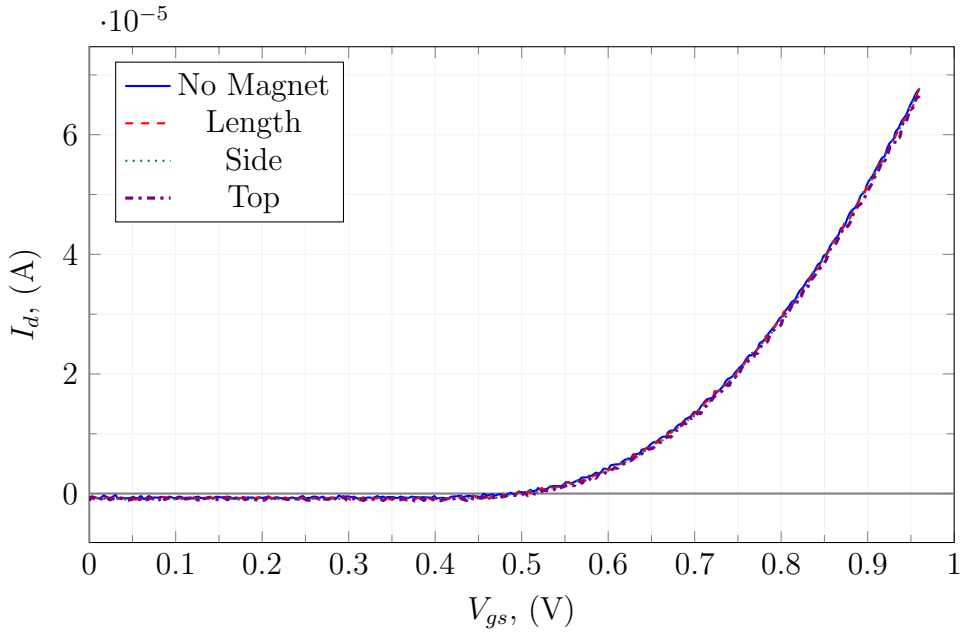


Figure 5.23: Zoomed output characteristics around the threshold.

Both the threshold voltage and the characteristics of the transistor were found to align with simulation results, showing little to no change when at field.

MD-Neuron Results

Once the NMOS test was complete, the MD-Neuron IC was also tested in each of the field orientations. Readings outside of the magnetic field were taken for reference. Each MD-Neuron IC contains two full neurons, and there are therefore two sets of results for each of the tests. The MD-Neuron IC was tested for its response to a full range of pulse and step stimuli. The IC chosen for this test was selected from the subset of functional ICs identified during the initial process variation tests to ensure meaningful results.

The results from the $100\mu s$, $49\mu A$ supra-threshold pulse is shown in Figure 5.24. There was no change in the neurons response when a strong magnetic field was applied in any orientation. It is possible to examine the models internal settling time constants by selecting the data from a stimulus close to the neurons internal threshold. In the case of Figure 5.25 a pulse of just $24\mu A$ was provided to each neuron. Due to the differences caused by process variation this stimulus caused one neuron to fire and the other to simply repolarise post-stimulus. This is an interesting result as it yields the neuron models behaviour at just either side of the membrane potential threshold.

Figure 5.25 shows that there was no change in the sub-threshold neuron. The supra-threshold neuron, however, does appear to perform differently for each of the field orientations. This difference in behaviour is not seen in the $23\mu A$ stimulus results, where neither neuron fires, or the $25\mu A$ stimulus results and is therefore likely due to this neurons threshold sitting very close to the $24\mu A$ stimulus. The boundary between AP generation and stimulus rejection is unstable meaning that models whose stimuli land upon this boundary can become sensitive to small fluctuations or noise, resulting in large changes to the models response. The fact that the models performs predictably either side of this stimulus supports this hypothesis.

The neurons response to step inputs was also recorded, allowing the generation of tonic spiking to be monitored. Any differences in AP timing will present as different spiking frequencies making it easy to identify any small changes that would otherwise go unnoticed. Figure 5.26 shows the results from a 10ms $49\mu A$ pulse. In these results it is clear that the two neurons operate consistently regardless of magnetic field presence in any orientation.

Discussion

Both the NMOS and Neuron magnet test results suggest that any effect a static high magnetic field may have on the AMS $0.35\mu m$ IC technology is minimal. Some

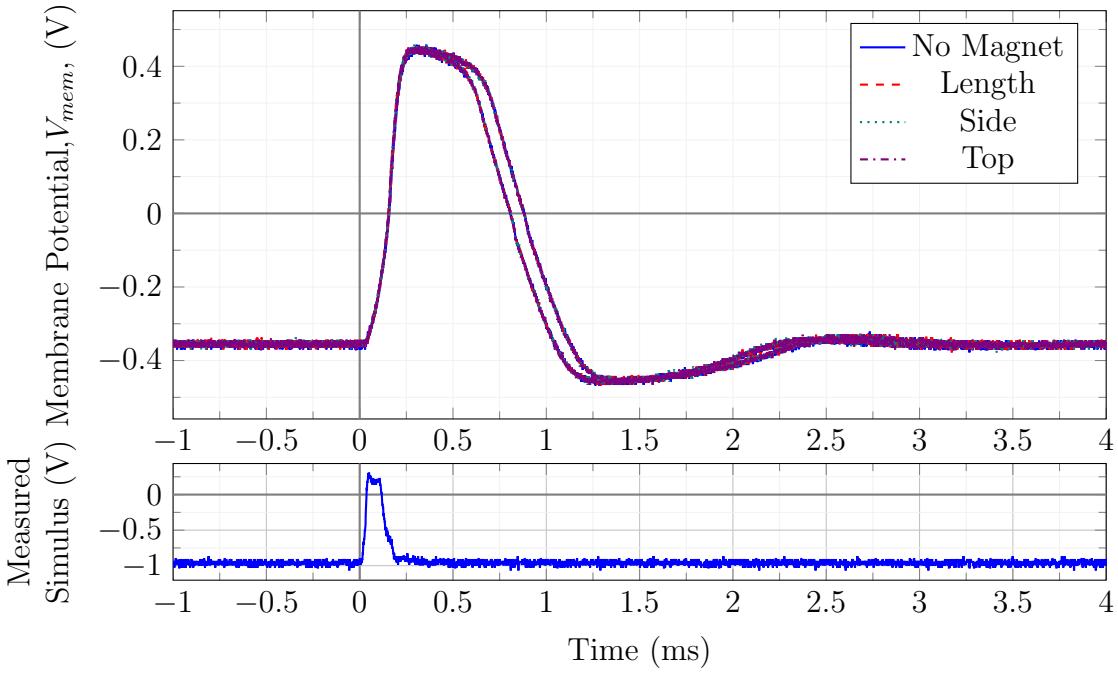


Figure 5.24: The MD-Neuron IC shows an identical response to a $100\mu s$ $49\mu A$ pulse stimulus both in and out of a static 3T magnetic field.

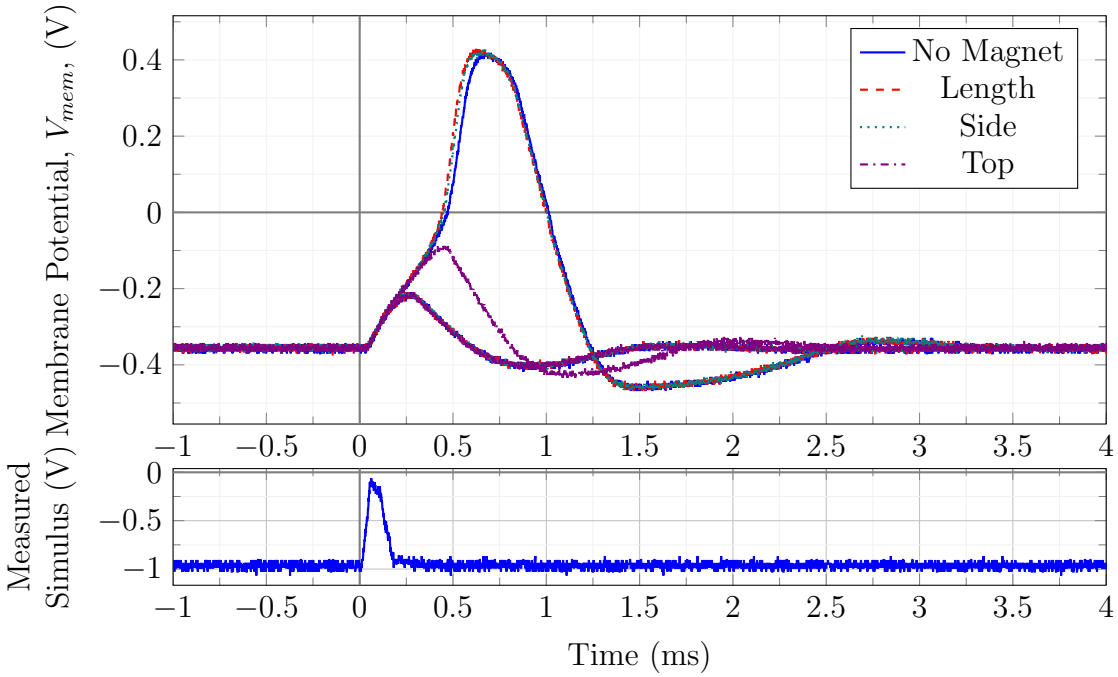


Figure 5.25: MD-Neuron IC membrane potential for a $100\mu s$ $24\mu A$ stimulus both in and out of field. One neuron within the IC is seen to fire while the other generates no AP, this difference is due to process variation.

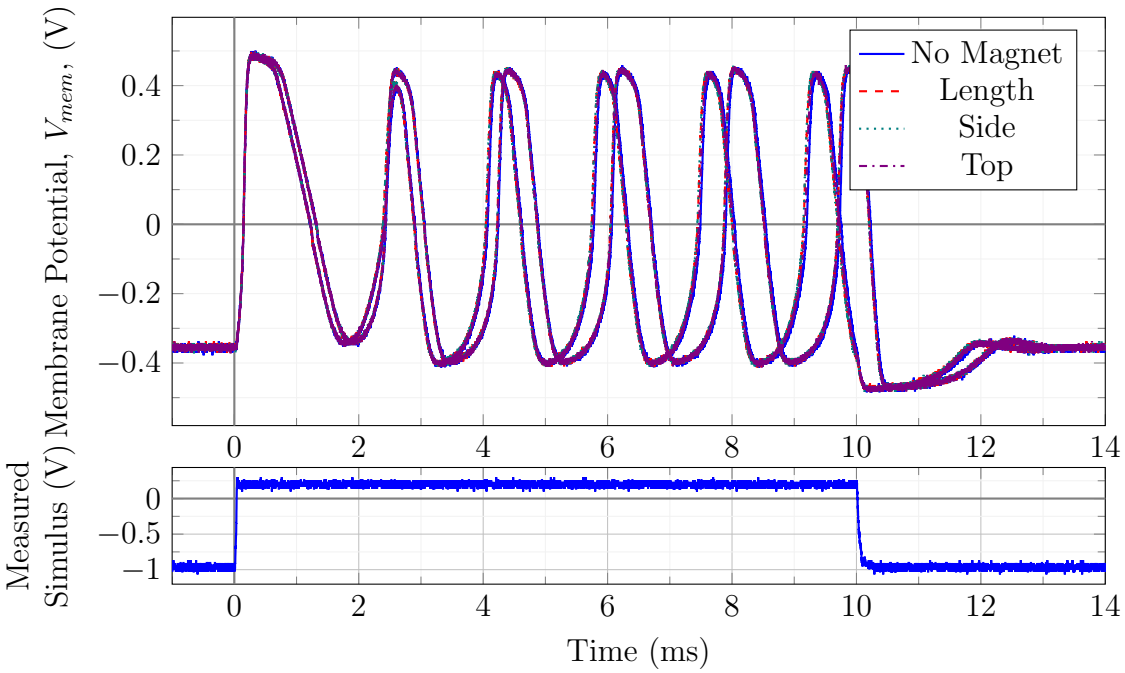


Figure 5.26: MD-Neuron IC membrane potential showing identical tonic spiking in response to a $10ms$ $49muA$ stimulus. In this test there is no discernable difference in the results of the in-field and out-of-field neurons.

small deviation from the expected value was seen in the NMOS V_{ds} curve traces for a magnetic field oriented through the top face of the IC. The scale of this difference seemed dependant on the V_{gs} value, with larger V_{gs} yielding a greater difference. At $V_{gs} = 1.50$v the change was approximately $2.5\mu A$ placing it two orders of magnitude smaller than the output signal itself. No other changes were observed in the NMOS results, as seen in Figures 5.20 and 5.22. Previous experiments performed by Hebrard *et. al.* found that transistor characteristics were seen to change in 7 to 10T fields, however the change was found to be proportional to the square of the field strength and deemed manageable within a 7T field [112]. This result aligns with the authors own experiments, where either negligible or no change was observed during the 3T field tests.

The lack of any change in the NMOS performance suggests that there should be little change in the Neuron IC itself. The results from the neuron tests, shown in Figures 5.24 and 5.26 support this assumption, showing no difference in the neurons AP response to a supra-threshold stimulus. The frequency of the tonic spiking remains constant, with the in-field APs appearing concurrently to that of the out-of-field results.

There was some small drift in the neurons operation for an at-threshold pulse, shown in Figure 5.25. These differences are likely due to the unstable nature of AP generation when provided with a stimulus that sits at the threshold level. In such a situation a small deviation caused by noise or other external factors can directly influence whether

an AP is generated or not. On inspection, the neurons were found to operate identically in- and out-of-field for stimuli at $\pm 1\mu A$ of this threshold level. The smaller stimulus resulted in neither neuron firing which directly supports the conclusion that the neuron was sitting at its threshold value.

The APs shown in Figures 5.24 and 5.25 bear some notable similarities and differences to the classical AP introduced in Figure 2.4. Firstly, the MD-neuron is seen to produce either Graded Potentials (GPs) or APs depending upon the stimulus. In the case of GPs, a sub-threshold depolarisation is seen under stimulus, with the neuron rapidly undergoing repolarization. As expected, this repolarization causes a slight overshoot beyond the resting potential. The AP response, however, appears to differ slightly in shape from that of the classical model. Many of the key properties of an AP are still apparent in Figure 5.25. Namely, the supra-threshold depolarisation may be identified before a sudden upstroke. The neuron then undergoes repolarization and hyperpolarization for a refractory period. The most notable difference is therefore in the peak shape of the AP itself. It has been shown that the internal timings of this model are strongly tied to both the parameter currents and process variation effects. This difference in peak shape is therefore most likely a direct result of the models parameters applied. Figure 5.16 also shows a clipped AP shape. In this case the model was operating with values close to that of the Sodium and Potassium Nernst potentials. This caused the neuron to limit the hyperpolarization and upstroke depolarisation resulting in a clipped appearance. Given that the general structure of the AP is maintained, these small fluctuation in AP morphology are not of primary concern.

The lack of any change in operation means that this MD-Neuron design is suitable for use in large static magnetic fields. The effects of process variation far outclass any effects caused by the presence of the magnetic field, as seen in the two on chip neurons - where each responds differently but consistently to the same input stimuli. The CResPace project must therefore explore methods to tune the neuron models post-fabrication to ensure that each and every neuron operates in a predictable and controllable manner. The design already includes bias currents to allow this tuning to take place, and the addition of simple but controllable current sources on chip would allow further exploration of each bias currents range and impact. Future studies into the impact of dynamic magnetic fields may now be performed in the knowledge that the large static field will cause no detrimental effect on the results or outcome.

5.3 Conclusions

Analogue IC design offers a direct, highly efficient and effective way to emulate biophysically accurate neural models with little hardware overhead. Despite this apparent solution, many new neuromorphic efforts heavily utilise digital methods within their hardware. The low adoption of analogue neural circuits may, in part, be

linked to the limited application of analogue ICs seen within other fields of electronics. Alongside this bias, a number of other barriers to mass adoptions have been identified within this work.

First, and foremost, analogue design requires considerable understanding of the underlying model and fabrication process. This dependence on process technology makes the generation of neuromorphic library parts impractical. Voltage rail levels and available floorspace adds further variation to the design and layout, with many factors also dependent on the interface requirements. This often results in the development of custom solutions for a given analogue neuron design on a given technology, representing a significant time commitment. In contrast, digital synthesis is a well established field that allows designers to share work. Due to developed digital pipelines and an inherent hierarchical structure, the technology underlying the implementation is often kept independent to the design itself making it easy to move to a new process with minimal effort. In general, the principles of analogue abstraction and hierarchy are recognised to differ significantly from that of digital design, meaning there is little computational framework when developing new analogue solutions. Many aspects of analogue abstraction remain to be discovered, with the general adoption of Turing machines and digital logic limiting the wider development and research of analogue design techniques [108].

Another limitation demonstrated by the current MD-neuron chip design is found in the use of large capacitors with each implemented neuron. The size of these capacitors is determined by the scale of the currents used within the model. Reducing the scale of the capacitors would require a reduction in the scale of the currents used throughout the circuit. To provide the relatively slow timings seen within biological neural function, biophysically accurate neural models must either use currents within the deep-sub-threshold region or larger capacitors operating as low-pass filters. Systems operating within the deep-sub-threshold region will be especially susceptible to noise and process variation, further adding to the complications and challenges associated with analogue design. Alternatively, small numbers of neurons may save space by utilising external capacitors, however large neural arrays become quickly constrained by the available IO resulting in a requirement for on-chip solutions. As seen in Figure 5.11a, if the currents are kept above the deep-sub-threshold region, these on-chip capacitors can dominate the available floorspace limiting the number of neurons that may be implemented upon a single IC.

Even when developing systems within the sub-threshold region and above, the effects of process variation cannot be ignored. While digital designs also suffer from process variation, its effect is minimised by the saturated mode of operation that digital systems utilise. In contrast, sub-threshold analogue designs can become highly susceptible to process variation resulting in changes from small fluctuations in system timings to outright model failures. Addressing and mitigating these effects is a time consuming task and often requires external bias values to be tweaked for each individual circuit instance. This would represent a significant time commitment and is not practical

when scaling the neural models to several thousand neurons.

This work has shown that the generation of small sets of biophysically accurate neural models is possible with analogue techniques. An implementation of an MD-neuron has been demonstrated, including both the design summary and validation results. Analogue designs are well suited for biophysically accurate neural models, offering considerable power efficiency and inherent non-linear operation in a manner which more closely reflects nature than that of digital systems. Despite these advantages, the development of practical and abstracted analogue solutions presents a significant design challenge. These challenges may be addressed with time, as analogue computational techniques and devices such as field programmable analogue arrays are further developed. With the wide adoption and established pipelines for digital design, however, it seems likely that digital models will remain the most common solution for large scale neuromorphic systems for many year to come. The remainder of this thesis considers such digital designs, looking at methods to optimise and accelerate these systems.

Chapter 6

Optimising Biophysically Accurate Neural Models

Chapter 5 introduced an analogue implementation of the Hodgkin-Huxley neuron model and identified a number of design challenges and complications when using this model in large scale neuromorphic systems. As a consequence of these design challenges many neuromorphic systems are developed using digital, rather than analogue, techniques. This allows designers to leverage well established digital design pipelines and build solutions that easily interface with existing processor technologies. Biophysically accurate models are often continuous, while digital systems are discrete. Some conversion must therefore be performed before the analogue neuron models may be implemented in digital systems.

A digital implementation of the Hodgkin-Huxley neuron, for example, requires the model to be first converted into the discrete time domain. In this chapter this conversion and associated design challenges are addressed in Section 6.2. Following this work the model must be optimised or approximated to allow efficient implementation. Section 6.3 considers the approximation of the exponential function, providing some important underlying representations of the operation. A number of approximations are then detailed in the remainder of this Section before they are compared and applied to the Hodgkin-Huxley model. Having identified suitable approximations for the exponential operation, a set of hardware implementations are generated using System Verilog. These implementations allow the size and speed of each approximation to be recorded and compared, as discussed in Section 6.4.

The hardware implementations use a library of reconfigurable floating point maths blocks, developed by Alex Beasley at the University of Bath. These blocks were chosen because they provided insight into their inner workings, ensuring that there was no hidden optimisations that would only be applied to some of the approximations under consideration.

6.1 Digital Numerical Representations

Digital systems are inherently quantized and discrete. Neuron models must therefore be converted to a quantised value and discrete time domain form before they can be implemented using digital hardware. The degree of quantisation depends upon the numerical representation used within the system. Integer representations yield the simplest hardware but are incapable of representing decimal values and therefore require any implemented models to be quantised into integer steps. This level of quantisation often results in the loss of precision or model dynamics and is rarely acceptable when modelling neurons.

A fixed point representation uses a defined number of fractional bits to represent real values. In this representation the level of quantisation is determined by the scale of the least significant bit, such that the quantisation interval, Q, is given as follows:

$$Q = 2^{-B} \tag{6.1}$$

where B is the number of decimal bits. As such, the number of decimal bits must be chosen to ensure that the required precision is provided. This means that the precision is constant over the whole represented range, even if the modelled system only requires high precision for small values - as is often the case in neuron models. Additionally, both the integer and fixed point representations have a range defined by the number of integer bits used. This means that wide buses are required for large ranges or precise values.

Floating point representation (as defined by the IEEE 754 standard) splits the stored bits into three distinct parts: the sign bit, the exponent and the mantissa. These three parts are then used to construct the stored value as demonstrated below with the two values -0.4275 and 125.

$$-0.4375 = \overbrace{-}^{\text{Sign}}\, 1.\overbrace{7500000}^{Mantissa} \times 2^{\overbrace{(1021}^{Exponent} - 1023)}$$

$$125 = \overbrace{+}^{\text{Sign}}\, 1.\overbrace{9531250}^{Mantissa} \times 2^{\overbrace{(1029}^{Exponent} - 1023)}$$

In this case, the 1023 subtracted from the exponent is the bias, defined according to the double precision IEEE 754 standard. Using this form, floating point numbers have a considerably large representation range. The precision or quantisation of floating point numbers is dependant on the size of the number as the decimal mantissa is scaled by the exponent. This means that large numbers are quantized into larger intervals, while

smaller numbers use smaller quantisation intervals. As a result floating point numbers can represent both small-signal and large-signal systems with a relative precision.

Floating point numbers are used extensively within commercial processors meaning that hardware accelerators are most compatible when supporting a floating point interface. Additionally, biophysically accurate models require highly precise small-signal dynamics as well as a large numerical range. Floating point numbers can therefore provide the desired precision for the small-signal dynamics without unnecessarily increasing the precision over the whole range. This helps reduce the number of bits required to represent the full operational range. For these reasons the digital systems considered in this chapter will be developed as floating point solutions.

6.2 Digital Hodgkin-Huxley Models

For ease of reference the Hodgkin-Huxley model is restated below in Equation set 6.2, 6.3 and 6.4. In this model, V is the membrane potential; C_M is the membrane capacitance; I is the injected current; E_K, E_{Na} and E_L are the potassium, sodium and leakage Nernst potentials; n and m are the potassium and sodium activation gating variables; h is the sodium inactivation gating variable; and $\bar{g}_K$, $\bar{g}_{Na}$ and $\bar{g}_L$ are the maximum potassium, sodium and leakage conductances. Equation 6.2 is termed the Hodgkin-Huxley equation; Equation set 6.3 are the gating variable equations; and Equation set 6.4 are the rate equations.

$$C_M \frac{dV}{dt} = I - \overbrace{\bar{g}_K\ n^4(V - E_K)}^{I_k} - \overbrace{\bar{g}_{Na}\ m^3 h(V - E_{Na})}^{I_{Na}} - \overbrace{\bar{g}_L(V - E_L)}^{I_L} \tag{6.2}$$

$$\begin{aligned} \frac{dn}{dt} &= \alpha_n(V) \cdot (1 - n) - \beta_n(V) \cdot n \\ \frac{dm}{dt} &= \alpha_m(V) \cdot (1 - m) - \beta_m(V) \cdot m \\ \frac{dh}{dt} &= \alpha_h(V) \cdot (1 - h) - \beta_h(V) \cdot h \end{aligned} \tag{6.3}$$

$$\begin{aligned} \alpha_n(V) &= 0.01 \frac{10 - V}{e^{\frac{10-V}{10}} - 1} & \beta_n(V) &= 0.125 e^{\frac{-V}{80}} \\ \alpha_m(V) &= 0.1 \frac{25 - V}{e^{\frac{25-V}{10}} - 1} & \beta_m(V) &= 4 e^{\frac{-V}{18}} \\ \alpha_h(V) &= 0.07 e^{\frac{-V}{20}} & \beta_h(V) &= \frac{1}{e^{\frac{30-V}{10}} + 1} \end{aligned} \tag{6.4}$$

Equations 6.2 to 6.4 represent a continuous-time analogue model and are therefore unsuitable for direct implementation using digital logic, which operates in the discrete time domain. The analogue model may be converted to a more suitable form using Euler's method of approximation [113]. The process to achieve this conversion is detailed in Appendix B. Once converted to the discrete time domain the Hodgkin-Huxley model may be directly implemented using the following steps:

1. Set the initial simulation values, $V_0 = V_{rest}$, $n_0 = 0$, $m_0 = 0$, $h_0 = 0$ and $j = 0$.

2. Calculate the α and β gating variables for each channel using Equation set 6.4.

3. Calculate the time constant, τ_x, and steady state value, x_{ss}, for n, m and h using:

$$\begin{aligned} x_{ss} &= \frac{\alpha_x}{\alpha_x + \beta_x} \\ \tau_x &= \frac{1}{\alpha_x + \beta_x} \end{aligned} \tag{6.5}$$

4. Calculate the next gating probability (n, m and h) values using Euler's method:

$$\begin{aligned} n_{j+1} &= n_{ss}(t_j) - (n_{ss}(t_j) - n_j) \cdot e^{-\frac{\Delta t}{\tau_n}} \\ m_{j+1} &= m_{ss}(t_j) - (m_{ss}(t_j) - m_j) \cdot e^{-\frac{\Delta t}{\tau_m}} \\ h_{j+1} &= h_{ss}(t_j) - (h_{ss}(t_j) - h_j) \cdot e^{-\frac{\Delta t}{\tau_h}} \end{aligned} \tag{6.6}$$

5. Calculate the channel conductance, g_i, for each set of simulated ionic channels:

$$\begin{aligned} g_k &= \bar{g}_K \; n_j^4 \\ g_{Na} &= \bar{g}_{Na} \; m_j^3 h_j \\ g_L &= \bar{g}_L \end{aligned} \tag{6.7}$$

6. Calculate $\sum g_i$ and $\sum g_i E_i$.

7. Calculate the membrane potential time constant, τ_V, and steady state value, V_{ss}, using:

$$\begin{aligned} V_{ss} &= \frac{\sum g_i E_i}{\sum g_i} + \frac{I_{inj}}{\sum g_i} \\ \tau_V &= \frac{C}{\sum g_i} \end{aligned} \tag{6.8}$$

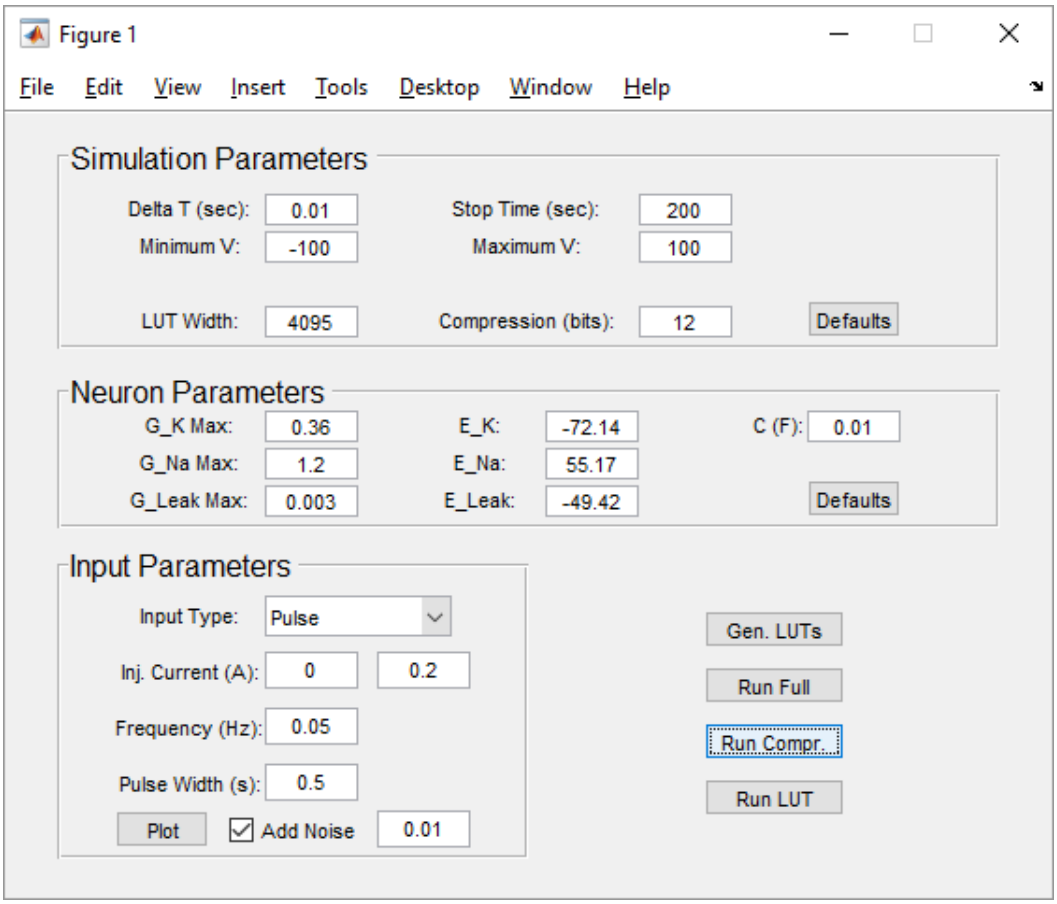


Figure 6.1: GUI for controlling the MATLAB Hodgkin Huxley simulator, showing the default neuron parameters. Input stimuli may be produced from a set of parametrised input shapes, with the option to add additional red noise to the signal.

8. Calculate the next membrane potential value, V_{j+1}:

$$V_{j+1} = V_{ss}(t_j) - (V_{ss}(t_j) - V_j) \cdot e^{-\frac{\Delta t}{\tau_V}} \tag{6.9}$$

9. Increment time index, j.

10. Repeat from Step 2.

This process was used to develop a Hodgkin-Huxley simulator in MATLAB. The simulator supports custom neuron parameters such as channel conductance, Nernst potentials and the membrane capacitance. Parametrised DC, step, pulse, ramp and sine input stimuli are supported, with the option to include red noise distortion on the input signal. The entire simulation system is controlled using a Graphical User Interface (GUI), shown in Figure 6.1.

The gating probabilities and resulting membrane potential for both a noisy pulse and sine stimulus are shown in Figures 6.2 and 6.3 respectively. These figures show

the system generating the characteristic Action Potential (AP) when excited by a sufficiently large input stimulus. The first 10ms of data in each simulation must be disregarded as the system has a transient response while the internal variables find a stable equilibrium.

6.2.1 Computational Cost

The Hodgkin-Huxley model is a computationally expensive model to implement. This is especially apparent in the discrete numerical representation, where four different Ordinary Differential Equations (ODEs) must be computed for each simulated time-step. Computational time is the key constraint when utilising this model, thus any methods that reduce the computational time of a each time-step have compound effects on the overall simulation speed.

The model makes considerable use of both the division and exponential operations. These operations are computationally expensive, as shown in Figure 6.4, and often require estimation to provide rapid calculation. Before resorting to approximations, however, it is possible to first rephrase parts of the model into a more computationally efficient representation.

In Equation 6.9 the exponential power contains a division, with the variable τ_V as the denominator. This τ_V variable is defined in Equation set 6.8 and also contains a division where the time-dependant variable forms the denominator. By recognising that τ_V is only used in Equation 6.9, it may be redefined or even incorporated into the final Equation as follows:

$$V_{j+1} = V_{ss}(t_j) - (V_{ss}(t_j) - V_j) \cdot e^{-\frac{\Delta t}{C} \sum g_i} \tag{6.10}$$

In this form, $-\Delta t/C$ is constant, making it possible to compute this value at the start of the simulation. This removes the requirement for any division within this operation, replacing it instead with a single multiplication. These methods do not add any error to the final calculation, as the two equations are mathematically equivalent, however multiplication is easier to perform making the final solution more efficient. This highlights the importance of the implementation methodology when investigating the computational expense of any process. As a further example of this principle, the impact of implementation approach may also be seen in Infinite Impulse Response (IIR) filter design. IIR systems of Nth-order are often implemented using one of two distinct methods. The first, termed direct form I, is the most straightforward design requiring 2N delay elements, while the second, termed direct form II, requires N delay elements. By changing the order of operations in the implementation of the filter, direct form II achieves the same representation with half the required delay elements of direct form I.

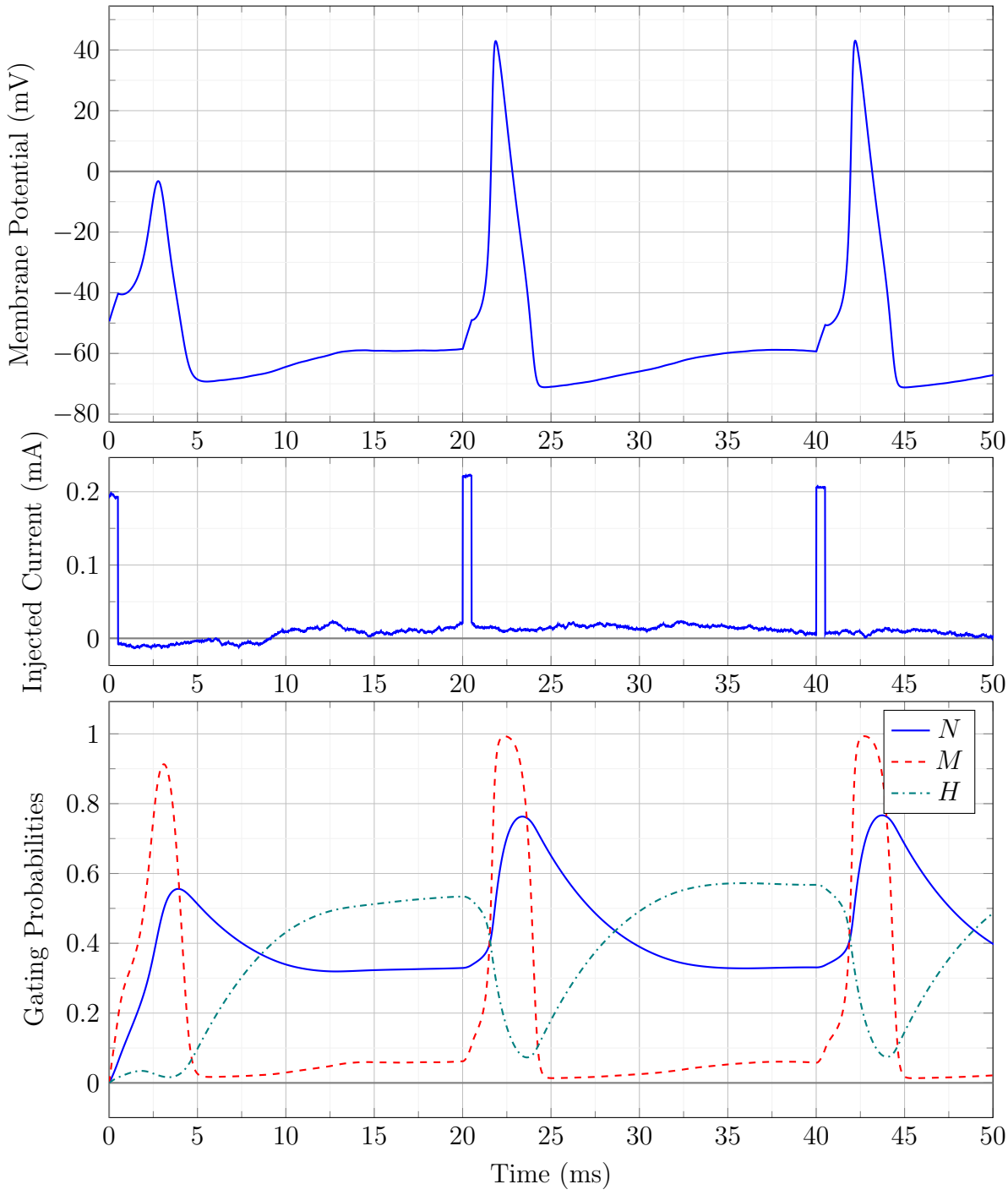


Figure 6.2: Outputs from the MATLAB Hodgkin Huxley simulator, using a 0.5ms, $0.2\mu A$ injected pulse stimuli with noise generation enabled. The membrane potential plot shows three distinct APs generated in response to the input stimuli, with the internal gating probabilities, N, M and H, responsible for generating these APs shown.

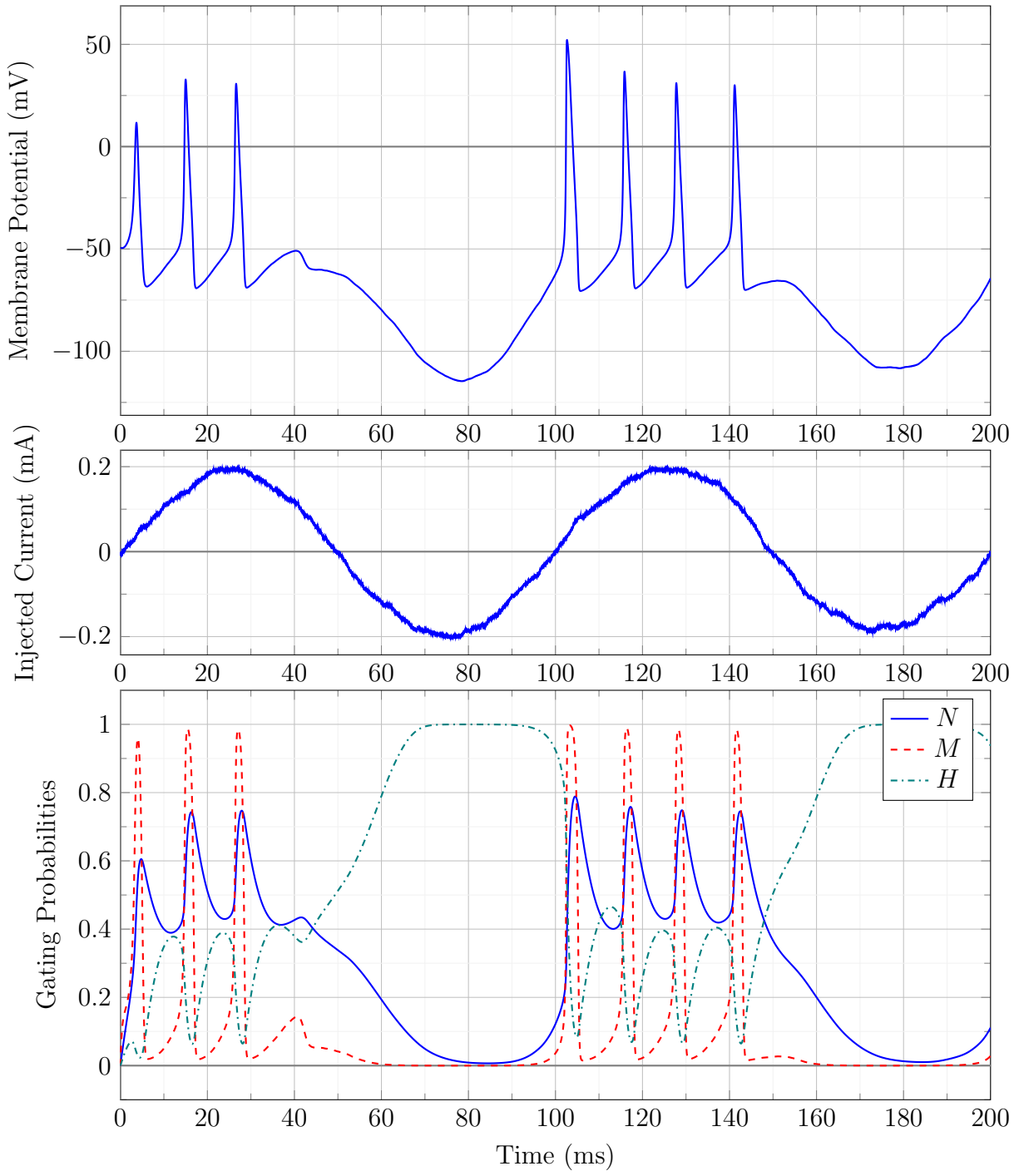


Figure 6.3: Outputs from the MATLAB Hodgkin Huxley simulator, using a 10Hz, $0.2\mu A$ sinusoidal stimuli with noise generation enabled. The membrane potential plot shows tonic spiking APs generated when the input stimuli exceeds the neurons threshold, with the internal gating probabilities, N, M and H, responsible for generating these APs shown.

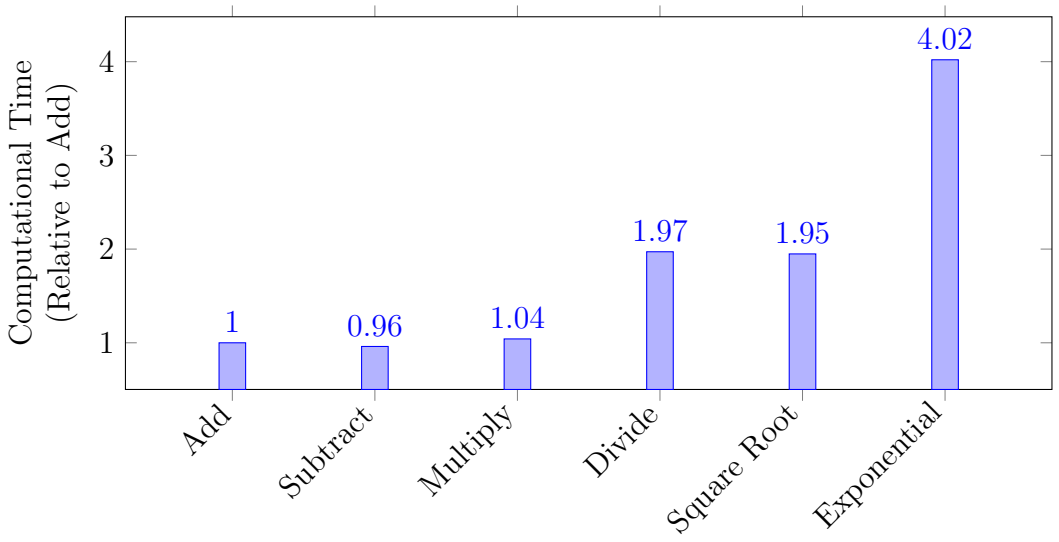


Figure 6.4: Computational time for each operation on a standard processor, shown relative to the Addition operation. These benchmark measurements were taken on an Intel i7-8700K.

Manipulating the operations in this way is not always possible and there are cases where approximations are necessary to provide the desired speed or efficiency. The exact nature of the approximations will depend on whether the solution is area or speed constrained. The reciprocal and exponential functions are two elements widely identified as computationally challenging operations [114, 115]. This is especially apparent when considering Field Programmable Gate Array (FPGA) implementations of neural networks, where the exponential and division operations are seen to use the largest number of resources in Look-Up Table (LUT), Flip Flop (FF) and Digital Signal Processing (DSP) blocks [116]. It is necessary to explore different implementation techniques for efficient operations if large numbers of biophysically accurate neuron models are to be implemented effectively in digital hardware. The remainder of this chapter considers implementations of the exponential function, looking at the impact and potential gains caused by utilising different approximations.

6.3 Approximating Exponentials

Having identified that exponential and division operations represent the highest computational cost within the Hodgkin-Huxley model, this section will now consider ways to approximate the exponential function. When approximating any function the required speed, resource, and accuracy will largely determine the suitability of the implementation. With floating point numbers, operation error is often measured relative to the Unit in the Last Place (ULP), which is the smallest value represented by the least significant bit in the mantissa. In the case of the Hodgkin-Huxley model it

could be argued that the accuracy must be to 1 ULP to ensure that the approximation matches the accuracy of the original function. While this is possible, it results in large and slow approximative models. In this section it is assumed that some small approximation error is acceptable, so long as the neuron model operates in a functionally equivalent way to that of the original model. This is a valid assumption if the desire is to run multiple large scale neural simulations - exploring a wide range of parameters and setups.

Choosing a Suitable Base

Digital systems typically utilise binary representations and operations in base-2 are therefore often more computationally efficient than operations in other bases. This is especially apparent when computing integer powers, where powers of two may be computed using a simple bit-shift operation. For this reason it is beneficial if the natural exponent is converted into a power of two by first defining a new converted input, n, such that:

$$e^x = 2^n \tag{6.11}$$

Taking the natural log of both sides yields:

$$x = ln(2^n) = n \cdot ln(2) \tag{6.12}$$

Rearranging this equation provides the conversion factor required to calculate n, as follows:

$$n = \frac{1}{ln(2)} \cdot x \tag{6.13}$$

This converted input allows all exponential operations to be replaced with powers of two, resulting in operations that are more easily implemented within binary systems.

Fractional Exponentials

Integer powers of two may be calculated through simple bit-shift operations, as shown previously. Fractional powers, however, are more complicated and often require approximation to meet performance requirements. The supported interval over which

the approximation must be valid may be reduced by first isolating the integer and fractional parts of the operation using the following relationship:

$$2^n = 2^{\lfloor n \rfloor} \cdot 2^{frac(n)} \tag{6.14}$$

where n is a real value and $frac(n)$ is the fractional part of n, defined as $frac(n) = n - \lfloor n \rfloor$. This enables the integer part to be computed using a bit-shift operation, while the fractional part may be calculated by some approximative means. Defined in this way, $frac(n)$ falls within the positive range $frac(n) \in [0, 1]$. This limits the range for the approximation, constraining and focusing the approximation effort while improving resource utilisation.

The remainder of this Section considers a number of approximations for e^x and 2^n where $n \in \mathbb{R}\,[0, 1]$. In the case of 2^n approximations, each may be used in conjunction with the methods above to approximate the exponential function, defined mathematically as:

$$e^x = 2^{\left\lfloor \frac{1}{ln(2)} \cdot x \right\rfloor} \cdot f\left(\frac{1}{ln(2)} \cdot x - \left\lfloor \frac{1}{ln(2)} \cdot x \right\rfloor\right) \tag{6.15}$$

where $f(x)$ is the chosen approximation of 2^x for the range $x \in [0, 1]$.

6.3.1 Linear Interpolation

Piecewise linear interpolation is arguably the simplest of approximative methods. This method uses linear polynomials to approximate a function as a set of linear regions. This is demonstrated in Figure 6.5, where two lines are used to provide a coarse approximation of the function $y = 25^x$.

There is significant error visible in the approximation shown in Figure 6.5. Whether this error is acceptable depends upon the end application. In cases where greater precision is required, designers may choose to increase the line count by dividing the function into successively smaller regions. These regions need not be regular, and the method of selecting suitable boundaries will largely dictate the associated error of the approximation. While each additional line will increase the accuracy of the approximation, it will also require additional memory or compute resource to store and calculate the new linear region. There is a trade-off between resource requirement and approximation accuracy when selecting the number of linear regions.

The approximation shown in Figure 6.5 represents a simplistic piecewise linear model, where each line directly connects the data points located at the region boundaries. Referred to as point-to-point piecewise approximations in this work, this approach

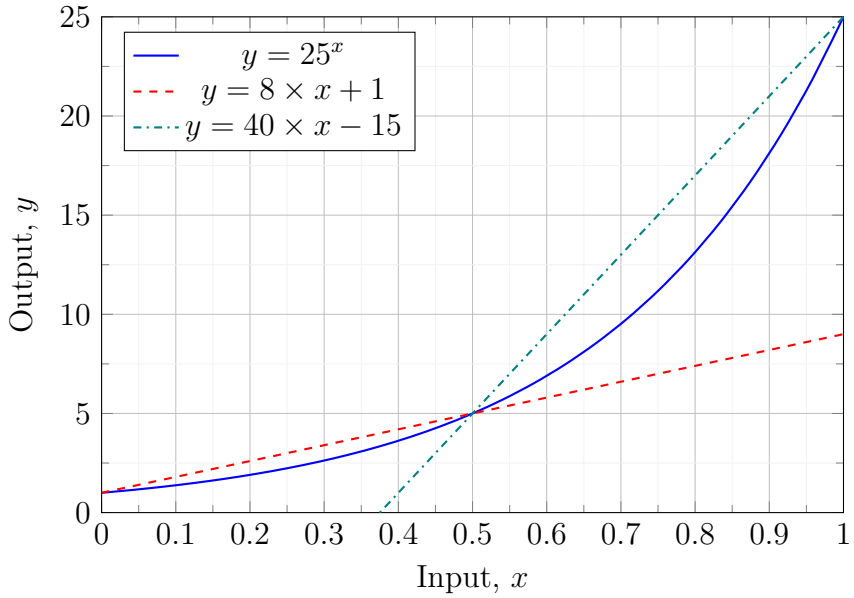


Figure 6.5: Demonstration of a simple 2-line piecewise linear approximation for the function $y = 25^x$.

means that there is zero error at the boundaries, with the largest error located near the centre of the region, as shown in Figure 6.6. It is possible to spread this error across the valid region by carefully adjusting the lines gradient and offset. The methodology behind this correction depends on whether the maximum absolute error or maximum percentage error must be minimised.

Calculating the Maximum Absolute Error

Each line in the point-to-point piecewise approximation may be defined in the following form:

$$l_n = a_n x + b_n \tag{6.16}$$

where a_n is the gradient and b_n is the offset of the n-th line, l_n. The gradient and offset values are defined by the linear interpolation of the two boundary points (x_1, y_1) and (x_2, y_2) across the valid region and may be calculated as follows:

$$\begin{aligned} a &= \frac{y_2 - y_1}{x_2 - x_1} \\ b &= y_2 - a x_2 \end{aligned} \tag{6.17}$$

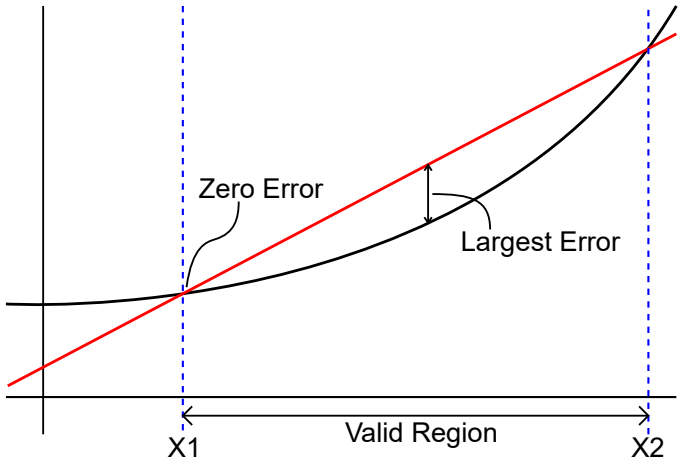


Figure 6.6: Example of the absolute error for point-to-point piecewise linear approximation when applied to the function $y = 2^x$ between two boundary points, X_1 and X_2.

The absolute error for an approximation of $y = 2^x$ may be defined within each valid region as:

$$E_{Abs} = |ax + b - 2^x| \tag{6.18}$$

Taking the differential of this error with respect to the input, x, yields:

$$\frac{dE_{Abs}}{dx} = \frac{ax + b - 2^x}{|ax + b - 2^x|} \cdot (a - 2^x ln(2)) \tag{6.19}$$

This differential is undefined at $ax + b = 2^x$, where the error is zero. By definition, this occurs at the boundary points in the point-to-point piecewise linear approximation. Solving for $dE_{Abs}/dx = 0$ then provides the following location for the maximum absolute error, x_{AbsErr}:

$$x_{AbsErr} = log_2\left(\frac{a}{ln(2)}\right) \tag{6.20}$$

This function is independent of the offset value, b, meaning that the maximum absolute error position will remain constant regardless of whatever line offset value, b, is used. The location of this maximum error may be used in Equation 6.18 to find the magnitude for the maximum error, E_{AbsMax}:

$$E_{AbsMax} = alog_2\left(\frac{a}{ln(2)}\right) + b - \frac{a}{ln(2)} \tag{6.21}$$

This maximum error is for the point-to-point piecewise approximation and is therefore not optimised in any way.

Minimising the Maximum Absolute Error

A corrective offset, c, may be added to the line in effort to reduce the maximum absolute error. In this case, the line takes the form $l_n = a_n x + b_n + c_n$, and the error may be defined as follows:

$$E_{AbsC} = |ax + b + c - 2^x| \tag{6.22}$$

As before, the position of the maximal error will be independent of the offset value. From observation it may be seen that shifting the line down to reduce the error at the maximum error point will also result in increasing error at the two boundary locations. It may be shown that the minimum maximal error will occur when $E_{x_1} = E_{x_2} = E_{AbsMaxC}$. While the corrective offset may be found mathematically, as shown in Appendix C, it may also be seen from Figure 6.6 that shifting the line down by half the maximum error will reduce the error at the maximum error point by half, while increasing the error at the boundary points by the same amount. This means that the final corrective value to achieve minimal maximum error is given by:

$$c = -\frac{1}{2} E_{AbsMax} \tag{6.23}$$

This yields a final line of the form:

$$l_n = a_n x + b_n - \frac{1}{2} E_n \tag{6.24}$$

where a_n is the gradient, b_n is the initial offset and E_n is the maximum error, E_{AbsMax}, for the n-th line, defined in Equations 6.17 and 6.21. While this is a relatively simple corrective measure to employ, there is a considerable limitation in its practicality. Originally, two adjacent boundaries shared the same boundary point, ensuring that the approximation was continuous. With the corrected model, each region employs its own corrective value resulting in discontinuity between regions. Any gradient descent operation performed on this approximation will therefore risk becoming stuck at the region boundaries.

Minimising the Maximum Percentage Error

While absolute error is easy to calculate, its application in exponential functions can result in unnoticed but significant error for small input values. Percentage error may therefore yield a more suitable metric for such functions as it incorporates the target value within its formation. Unlike the absolute error case, shifting the approximation offset, b, will result in different magnitudes of percentage error at the two boundary locations. For this reason it is necessary to modify both the gradient, a and offset, b, when minimising the percentage error.

The percentage error between the lines $l = ax + b$ and $y = 2^x$ may be defined as follows:

$$E_{\%} = \frac{l - y}{y} \tag{6.25}$$

where y is the actual output of the exponential 2^x.

As before, it may be assumed that the minimal maximum error will occur when the error at the valid region boundaries (x_1 and x_2 in Figure 6.6) is equal to the largest error within the valid approximation region itself. Under this assumption, a gradient-offset scale factor k may defined using the two boundary points, such that $a = kb$, as follows

$$\frac{(ax_1 + b) - y_1}{y_1} = \frac{(ax_2 + b) - y_2}{y_2}$$

$$y_2((ax_1 + b) - y_1) = y_1((ax_2 + b) - y_2)$$

$$ax_1y_2 + by_2 - y_1y_2 = ax_2y_1 + by_1 - y_1y_2$$

$$a(x_1y_2 - x_2y_1) = b(y_1 - y_2)$$

$$a = b\frac{y_1 - y_2}{x_1y_2 - x_2y_1} = kb$$

$$\therefore k = \frac{y_1 - y_2}{x_1y_2 - x_2y_1} \tag{6.26}$$

This gradient-offset scale-factor may then be used to redefine the linear approximations with a single parameter, b. This scaling will provide the required gradient to ensure that the percentage error at the two boundaries will be equal for any selected offset

value.

$$l = kbx + b \tag{6.27}$$

Using Equation 6.27 in Equation 6.25, the derivative of the percentage error may be shown to be:

$$\frac{dE_{\%}}{dx} = -2^{-x}b\left(kxln(2) - k + ln(2)\right) \tag{6.28}$$

As before, the maximum error will occur at the roots of Equation 6.28, defined as follows:

$$\begin{gathered} b = 0 \\ k \neq 0,\ x = \frac{k - ln(2)}{kln(2)} \end{gathered} \tag{6.29}$$

The first of these roots may be ignored since $b = 0$ would result in a zero gradient line that sits along the axis. The maximum error therefore occurs at the following location:

$$x_w = \frac{k - ln(2)}{kln(2)} \tag{6.30}$$

With the worst error located, it is once again possible to spread the error by setting the offset, b, such that $E_{\%w} = -E_{\%x_1}$:

$$\begin{aligned} E_{\%w} &= -E_{\%x_1} \\ \frac{kbx_w + b - y_w}{y_w} &= \frac{kbx_1 + b - y_1}{y_1} \\ kbx_w y_1 + by_1 - y_w y_1 &= kbx_1 y_w + by_w - y_1 y_w \\ b\left(kx_w y_1 + y_1 + y_w + kx_1 y_w\right) &= 2y_1 y_w \\ \therefore b &= \frac{2y_1 y_w}{kx_w y_1 + y_1 + y_w + kx_1 y_w} \end{aligned} \tag{6.31}$$

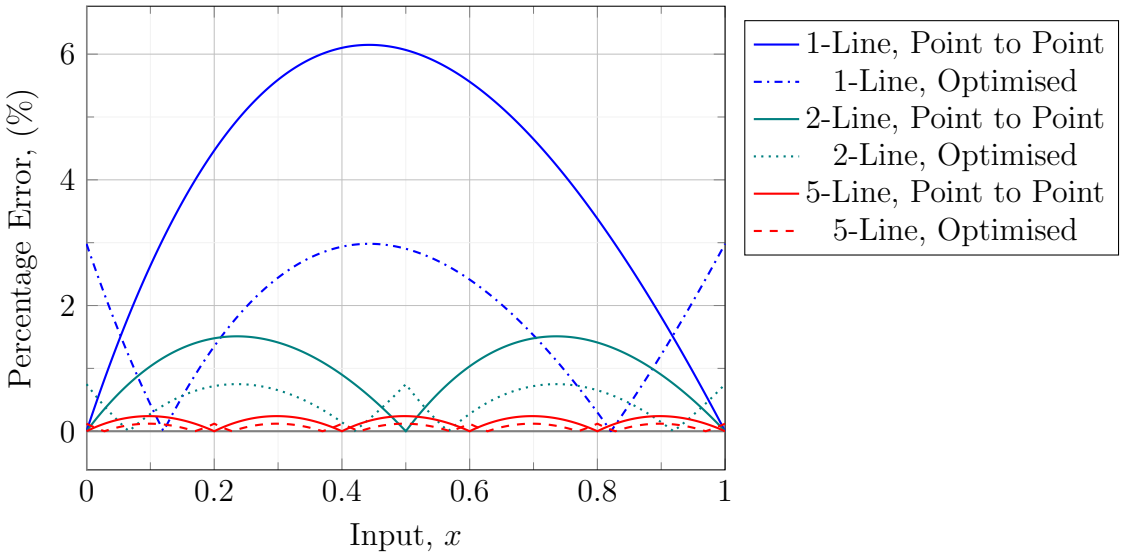


Figure 6.7: The percentage error for point-to-point and percentage error optimised piecewise linear approximations of the line $y = 2^x$. Showing decreased percentage error in the optimised model for 1-, 2-, and 5-line approximations.

Brought together, Equations 6.26, 6.27, 6.30 and 6.31 provide a method for finding the optimal piecewise linear approximations of the function $y = 2^x$. Figure 6.7 shows the percentage error for 1, 2 and 5 line piecewise approximations of 2^x, with and without this optimisation. From this figure it is clear that the optimised method achieves the minimum possible percentage error across the full range $x \in [0, 1]$ for any n-line piecewise linear approximation. Were the line to use a different gradient or offset the error at either the boundaries or max point would increase, resulting in an increased maximum percentage error.

The percentage error approximation may also be shown to provide continuity across the boundaries of the linear regions. This makes such approximations suitable for gradient descent optimisation techniques commonly used in computational neural models. One key downside to this method of approximation is found in the fact that there is error about $x = 0$. Such error can result in small-signal errors propagating within the model.

6.3.2 Polynomial Approximation

Instead of dividing the target function into multiple approximation regions, as with piecewise linear approximation, it is also possible to improve the accuracy of a model by increasing the order of the approximation itself. Unlike piecewise linear approximation, polynomial approximation uses higher-order polynomials to approximate the whole function space. The approximation becomes more accurate as the order is increased, however higher order polynomials require greater computation resulting in either slower

Table 6.1: Polynomial approximations of $y = 2^x$ with their associated maximum percentage error across the range $x \in [0, 1]$. In each case increasing the order of the approximation is seen to reduce the error by more than an order of magnitude.

Polynomial Approximation Function	Maximum Error (%)
$0.3427x^2 + 0.6494x + 1.0038$	3.7522×10^{-1}
$0.0790x^3 + 0.2241x^2 + 0.6968x + 0.9998$	1.8700×10^{-2}
$0.0137x^4 + 0.0517x^3 + 0.2417x^2 + 0.6929x + 1.0$	7.2154×10^{-4}
$0.0019x^5 + 0.0089x^4 + 0.0559x^3 + 0.2401x^2 + 0.6932x + 1.0$	2.2714×10^{-5}

or lager implementation hardware.

Typically polynomial approximations are unsuitable for exponential functions due to the diverging error caused by the difference between an nth order polynomial and a continuous exponential (the reasons for this divergence are made apparent in the derivations $d(y^x)/dx$ and $d(x^y)/dx$). In this case, however, the approximation is constrained within the range $x \in [0, 1]$ and it may therefore be approximated using a low-order polynomial with relative accuracy.

Table 6.1 shows four different polynomial approximations from 2nd order to 5th order. With each increase in order, the percentage error may be seen to reduce by more than an order of magnitude. The cost of this improvement comes at two additional multiplications operations and an additional add operation for each order.

As with the piecewise approximation, this error may be minimised mathematically by spreading the error between the maximum error point and the boundaries ($x = 0$ and $x = 1$). The methods behind such optimisation becomes increasingly complicated with each increased order of approximation.

6.3.3 Euler Generalised Continued Fractions

Thus far, approximations that utilise the base-2 conversion have been considered. There are many methods that provide approximations for fractional exponents directly. One method is a continued fraction for e^x that can be obtained via an identity of Euler. This method uses an iterative model, meaning that the system may use the same hardware to iteratively generate an increasingly accurate approximation of the exponential.

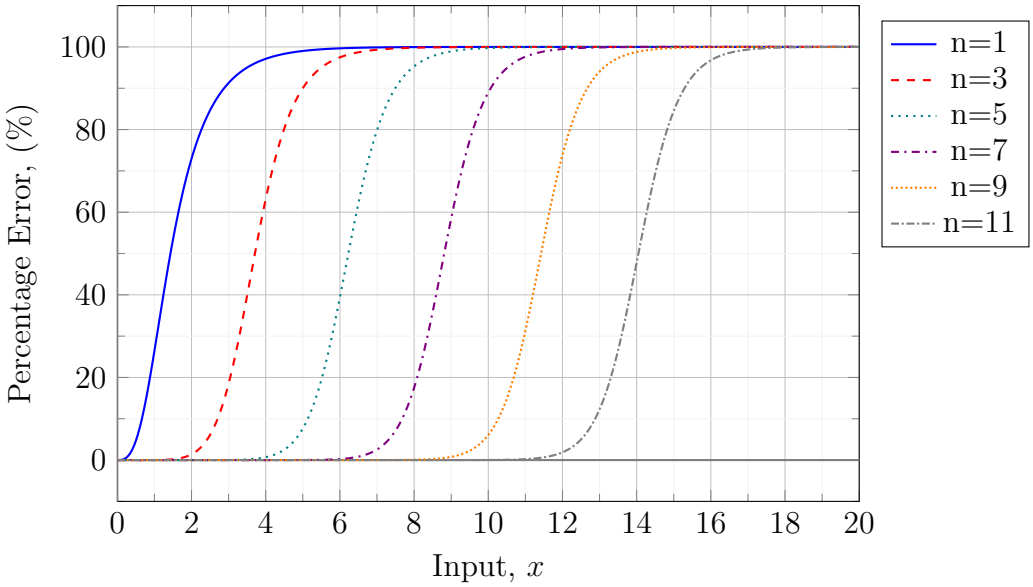
Mathematically, the continued fraction may be represented as either a single infinite

fraction or, as shown below, as a set of iterative functions:

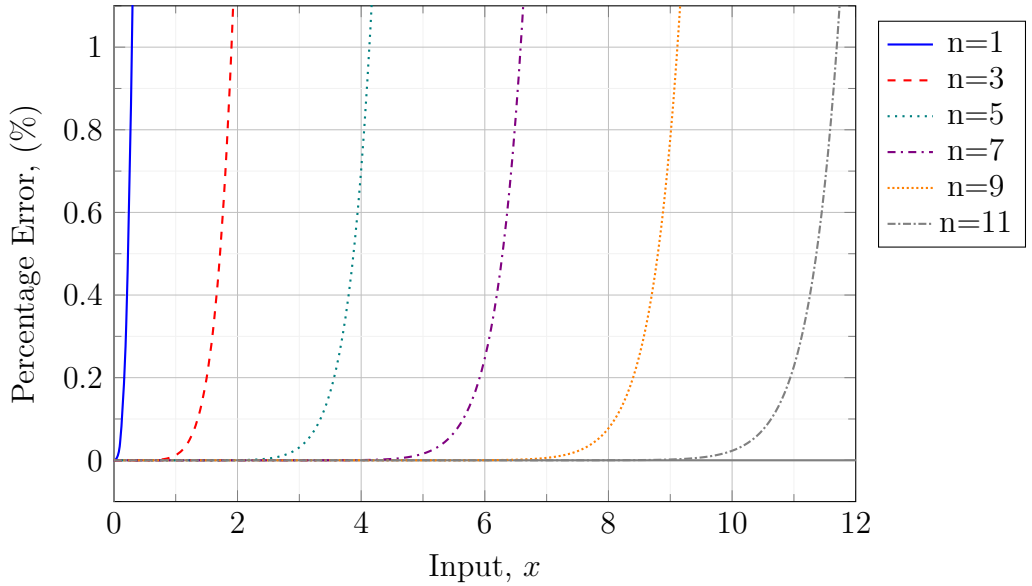
$$y = 1 + \frac{2x}{f(x,1) - x}$$
$$f(x,n) = 4n - 2 + \frac{x^2}{f(x,n+1)} \tag{6.32}$$
$$f(x,n_{limit}) = 1$$

In this form it is apparent how one block of hardware may be used to repeatedly compute the result of the function $f(x,n)$ until the desired accuracy is reached. In this way these implementations can provide the user with customisable accuracy post fabrication. The effect of increasing the depth, n, in this iterative model is seen in Figure 6.8, where each additional layer of iteration shifts the error curve along the input axis without greatly affecting the overall shape of the error function itself.

Considering the zoomed error plot shown in Figure 6.8b, it is clear that this approximation may be used to provide very-high accuracy for a known input range by selecting a suitable depth. It should noted that this method requires the depth to be select prior to calculation as the approximation must start at the chosen depth and propagate the result back up the chain. The downside of such iterative methods, however, is found in the latency of the system, where the iteration requirement results in an implementation that often takes more than n clock cycles to calculate a single output value. This latency scales quickly when computing many thousand operations within a simulation, resulting in a slow final solution. The act of approximating the exponential function with a large number of division operations also limits the realised efficiency of this model and is therefore only practical if an optimised division operation is already required, and yet available, elsewhere on the hardware.



(a) The full plot of the percentage error when approximating the exponential function e^x using different iterations, n, of the Euler continued fraction approximation model.



(b) A zoomed plot of the percentage error when approximating the exponential function e^x using different iterations, n, of the Euler continued fraction approximation model. The error is seen to rapidly cross the 1% point for each iteration depth, with deeper iterations providing a better range of valid approximations.

Figure 6.8: Plots of the percentage error caused by using Euler continued fraction models to approximate the exponential function e^x.

6.3.4 Power Series Approximation

Another common iterative method for approximating the exponential function may be achieved through the direct implementation of its power series representation. Defined mathematically below, the power series provides a means of calculating the result of a real exponential operation with increasing accuracy.

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{6.33}$$

Unlike the generalised continued fraction, the power series starts as a coarse approximation, refining its result with each additional element or iteration. This means that such implementations may be left running without strict timing requirements while other time consuming operations are performed. The error for this implementation is shown in Figure 6.9. As with the other iterative methods, this approximation provides very high accuracy for well constrained input ranges. While the power series still uses the division operation, the denominator is always an integer value meaning this approximation may utilise division hardware further optimised for such specific application.

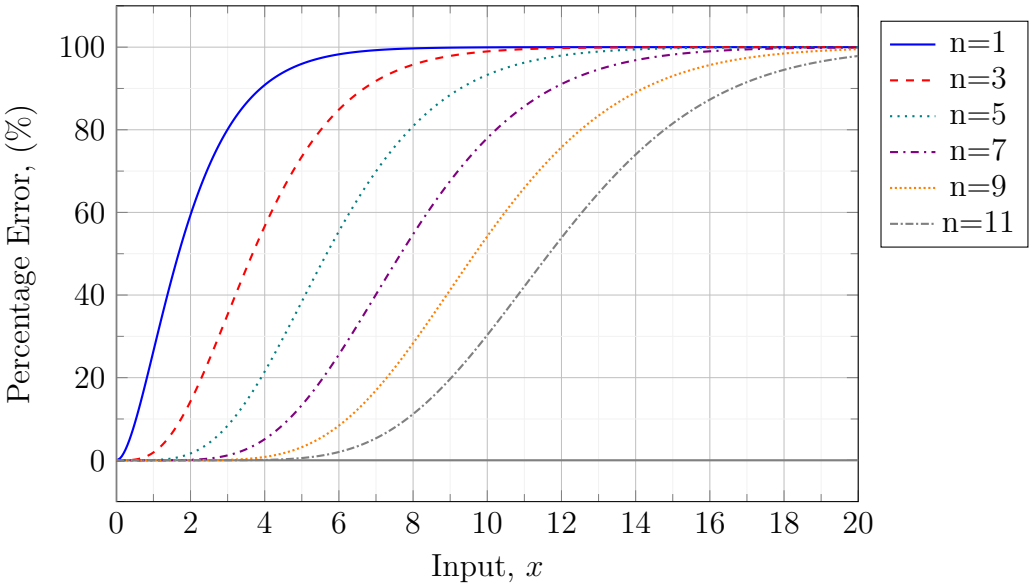
As seen in Figure 6.9, each additional layer of this iterative model both shifts the error and stretches the error function along the x axis. While this model may be further optimised for implementation in hardware, the error of this model climbs sooner than that of a continued fraction model of the same depth.

6.3.5 Small-signal Approximations

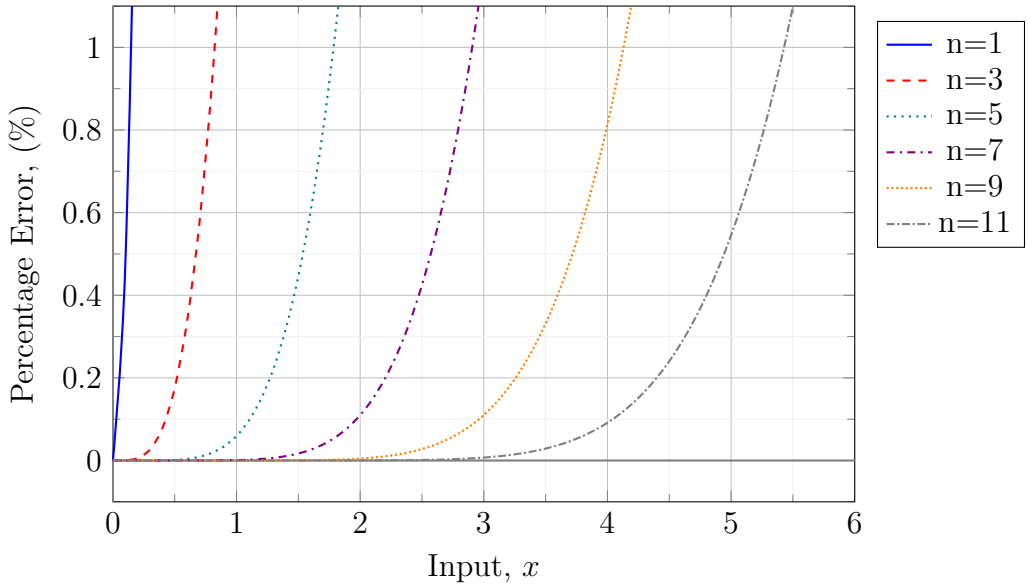
Each of the previously considered approximations have focused on accuracy first. A simplified model may be produced by instead designing an approximation focused on implementation efficiency. Starting with the small-signal approximation, it may be seen that e^x can be implemented with a single addition operation when operating on small values of x, as shown below:

$$e^x \approx 1 + x \tag{6.34}$$

This model is well suited for systems guaranteed to operate within the small-signal region where $x < 0.1$, however in the case of the Hodgkin-Huxley model the approximation must be valid for a wide range of input values. The high accuracy at values around the zero point, along with the simplicity of implementation makes this approximation a tempting alternative if the range may be constrained in some way.



(a) The full plots, showing the relatively rapid climb to 100% error for iteration counts from $n = 1$ to $n = 11$.



(b) Zoomed to show the error functions between 0 and 1% for different depths of iteration.

Figure 6.9: Plots of the percentage error caused by using the power series to approximate the exponential function e^x.

It has already been shown that a base-2 conversion constrains the required approximation range to between 0 and 1. In this way another implementation focused approximation may be produced such that:

$$2^x \approx 0.9645 + x \tag{6.35}$$

This approximation is significantly worse than other approximations considered within this chapter, with a maximum percentage error of 11.44%. However with a single addition operation, this error may be suitable in some conditions where the highly simplified implementation is of greater importance. The discontinuities introduced by this model are unavoidable due to the gradient being locked at 1.

The final issue seen with many of these approximations is the error about the origin. This may be addressed by actively switching the approximation model as the input values become small. For example, the simplified linear model shown above may be replaced with the small-signal approximation when the input is below a certain threshold. By setting this threshold at the crossover point of the two approximations it is possible to ensure continuity between the two approximations. This hybrid model uses the relationship $e^x = 2^{\lfloor n \rfloor} \cdot 2^{n-\lfloor n \rfloor}$ for most of the input values, switching to a more accurate representation only when suitable.

$$e^x \approx \begin{cases} 1 + x & \text{When } x < 0.0802, \\ (2 << \lfloor n \rfloor) \cdot (0.9645 + (n - \lfloor n \rfloor)) & \text{Otherwise} \end{cases} \tag{6.36}$$

where the $<<$ operator represents the left bit-shift operation that is equivalent to 2^n. The error for this model is shown in Figure 6.10 alongside the error of the individual elements that form the hybrid model.

Figure 6.10 shows how careful use of multiple approximation models can provide accuracy in the required operational regions without compromising the full approximation range itself.

6.3.6 Applying the Approximations

Having identified a number of different approximations, the impact of their application in the Hodgkin-Huxley model will now be considered. First the required operational range of the approximation must be found. Using the MATLAB simulator described in section 6.2, with the default parameters shown in Figure 6.1, it is possible to record the input values provided for each exponential operation. Figure 6.11 shows the values taken by each of the exponential operation inputs during a single AP under normal conditions. From this figure it is clear that a suitable approximation within the range $x \in [-10, 5]$ should be sufficient.

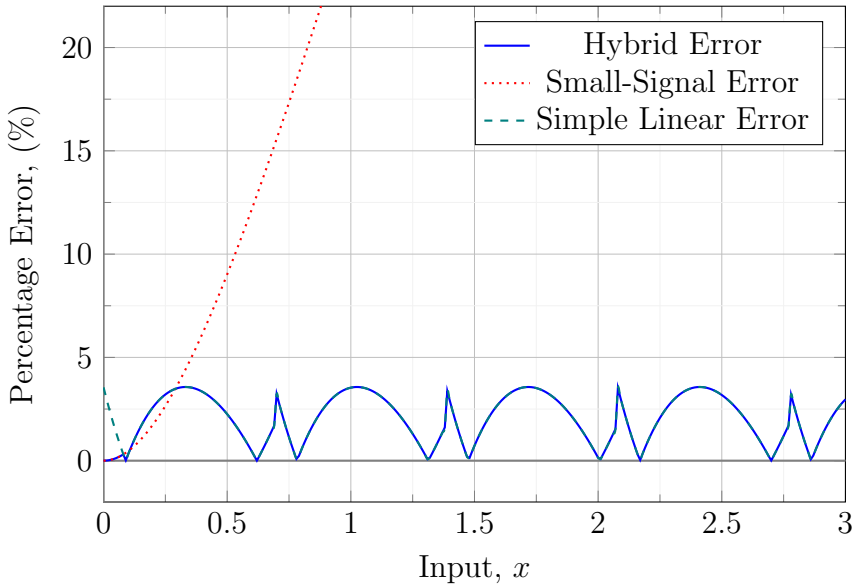


Figure 6.10: Percentage error for the hybrid model, showing how two different models may be combined to yield a model with carefully selected regions of high accuracy. In this model, the error about the origin is kept low, while the error across the entire range is constrained to within a constant maximum value of 3.57%.

It is possible to compute the maximum percentage error caused by an approximations application by evaluating the approximations error over the expected operational range. Table 6.2 shows this error for a set of incremental ranges between $[-20, 10]$. This limit was chosen as double the expected input range for the Hodgkin-Huxley model, as shown in Figure 6.11. Multiple observations about the approximations may be made from Table 6.2. Firstly, the models that utilise the power of two conversion (that is the linear piecewise and polynomial models) yield a constant maximal error regardless of the range over which they are applied. This makes them especially suited for applications requiring reliable approximation over large or undefined ranges. The downside of these models is found in their error at $x = 0$, meaning that these models may fail if the system utilising them is sensitive to small signals.

The Euler's continued fraction model results in large error for negative values, however this error may be reduced by increasing the iteration count. There is no error about $x = 0$, making this model very good for small-signal applications, even when using low iterations. The Euler's Continued Fraction is undefined for some values of x when using an even iteration count and the location of this node moves up the x-axis as the iteration count is increased.

The power series does not suffer from these undefined points. This said, the power series is intended for use with positive input values, resulting in very large error for negative values. As before, there is zero error at the origin, and when compared with Euler's continued fraction the error for small positive values is seen to decrease more rapidly as the iteration count is increased. Since the Hodgkin Huxley models exponential inputs

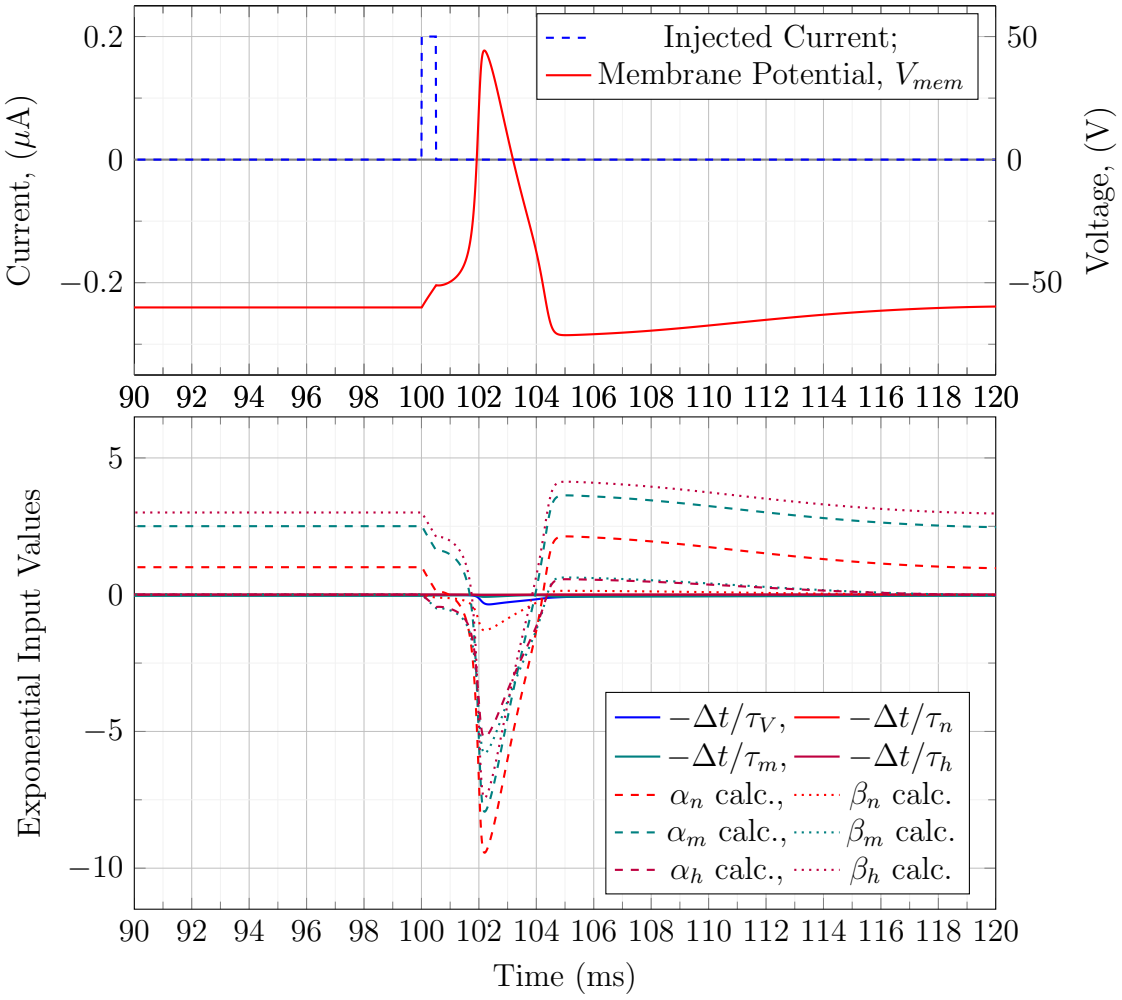


Figure 6.11: Exponential function inputs for the default neuron parameters when stimulated with a pulse input of $0.2\mu A$. From this plot it may be seen that a range of $[-10, 5]$ is required for an approximative exponential function to replicate the neural models original operation.

Table 6.2: Maximum percentage error for each approximative model when approximating the exponential function e^x over constrained ranges.

Approximation	Max percentage error (%) for input range 0 to...						
	-20	-15	-10	-5	0	5	10
1-Line Piecewise	2.98	2.98	2.98	2.98	2.98	2.98	2.98
2-Line Piecewise	0.75	0.75	0.75	0.75	0.75	0.75	0.75
3-Line Piecewise	0.33	0.33	0.33	0.33	0.33	0.33	0.33
4-Line Piecewise	0.19	0.19	0.19	0.19	0.19	0.19	0.19
2nd Order Polynomial	0.37	0.37	0.37	0.37	0.37	0.37	0.37
3rd Order Polynomial	0.02	0.02	0.02	0.02	0.02	0.02	0.02
4th Order Polynomial	0.00066	0.00066	0.00066	0.00066	0.00066	0.00066	0.00066
Euler Fraction, n = 1	4.39×10^{10}	2.86×10^{8}	1.81×10^{6}	1.01×10^{4}	0	99.02	99.99
Euler Fraction, n = 2	3.59×10^{10}	2.19×10^{8}	1.20×10^{6}	4.27×10^{3}	0	∞	∞
Euler Fraction, n = 3	2.41×10^{10}	1.28×10^{8}	5.39×10^{5}	901.20	0	90.01	99.98
Euler Fraction, n = 4	1.32×10^{10}	5.81×10^{7}	1.68×10^{5}	108.57	0	∞	∞
Power Series, n = 1	9.22×10^{11}	4.58×10^{9}	1.98×10^{7}	5.95×10^{4}	0	95.96	99.95
Power Series, n = 2	8.78×10^{12}	3.22×10^{10}	9.03×10^{7}	1.26×10^{5}	0	87.53	99.72
Power Series, n = 3	5.59×10^{13}	1.52×10^{11}	2.77×10^{8}	1.83×10^{5}	0	73.50	98.97
Power Series, n = 4	2.68×10^{14}	5.38×10^{11}	6.41×10^{8}	2.03×10^{5}	0	55.95	97.07
Power Series, n = 5	1.03×10^{15}	1.53×10^{12}	1.19×10^{9}	1.83×10^{5}	0	38.40	93.29
Power Series, n = 6	3.29×10^{15}	3.64×10^{12}	1.86×10^{9}	1.39×10^{5}	0	23.78	86.99
Hybrid	3.57	3.57	3.57	3.57	0	3.57	3.57

mainly settle in the negative region it is highly likely that the power series will fail to produce a suitable approximation.

Finally, the hybrid model offers the best of both worlds, with a constant maximal error across the full range of values and zero error at the origin. This constant error value may be reduced further by mixing different approximation models into the hybrid model as required.

It is not possible to asses these approximation models by error alone. The MATLAB simulator was therefore modified to incorporate each of these approximations, allowing their effects on the Hodgkin Huxley model to be seen directly. Figures 6.12 and 6.13 show the results of these implementations, displaying the membrane potentials depolarisation in response to a $0.5ms$ pulse input of $0.2\mu A$ and a $0.07\mu A$ step input respectively.

Using the results from this simulator it is possible to assemble a table of the approximation performance, considering the models capability to replicate three core elements in the Hodgkin Huxley model - the DC resting potential, phasic spiking and tonic spiking behaviours. The DC resting potential can be evaluated by how closely it matches the original models voltage. In this work, an error of less than 0.1% is deemed an acceptable approximation, an error of up to 1% a close approximation and anything greater as unacceptable. Phasic spiking is the term used to describe when a neuron generates a single AP in response to stimuli. If the timing of the AP, and its recovery, is closely matched to that of the original model the approximation may be deemed as good. If, however, the model successfully produces a single AP with different timings, the approximation may be considered close but not ideal. Finally if the model fails to produce a singular AP the approximation is unsuitable. Tonic spiking is seen when a neuron produces a fixed frequency of APs in response to a continuous input stimuli, in this mode any approximation that generates the same frequency of APs is considered a good match while approximations that present stable tonic spiking without matching the target frequency are only a close match. Models that fail to produce a stable tonic spiking response are considered unsuitable.

With these definitions, Table 6.3 has been assembled, showing the performance of each approximation method across the three core modes of operation. In this table, '✓' represents a good approximation, '≈' represents a close approximation, and '−' represents an unsuitable approximation.

Systems with relatively high percentage error, such as the 4 iteration Euler's continued fraction, are still capable of close representation of the Hodgkin Huxley model. At the same time systems with relatively small error, such as the linear piecewise models, struggle to accurately reproduce the tonic spiking behaviour seen in the original model. The power series model is locked in a continued fire mode and requires a total of 21 iterations before it produces a tonic spiking behaviour that is dependant on input stimulus. These results suggest that the success of the approximation model depends on a mixture of both a low full range error performance and very low small-signal

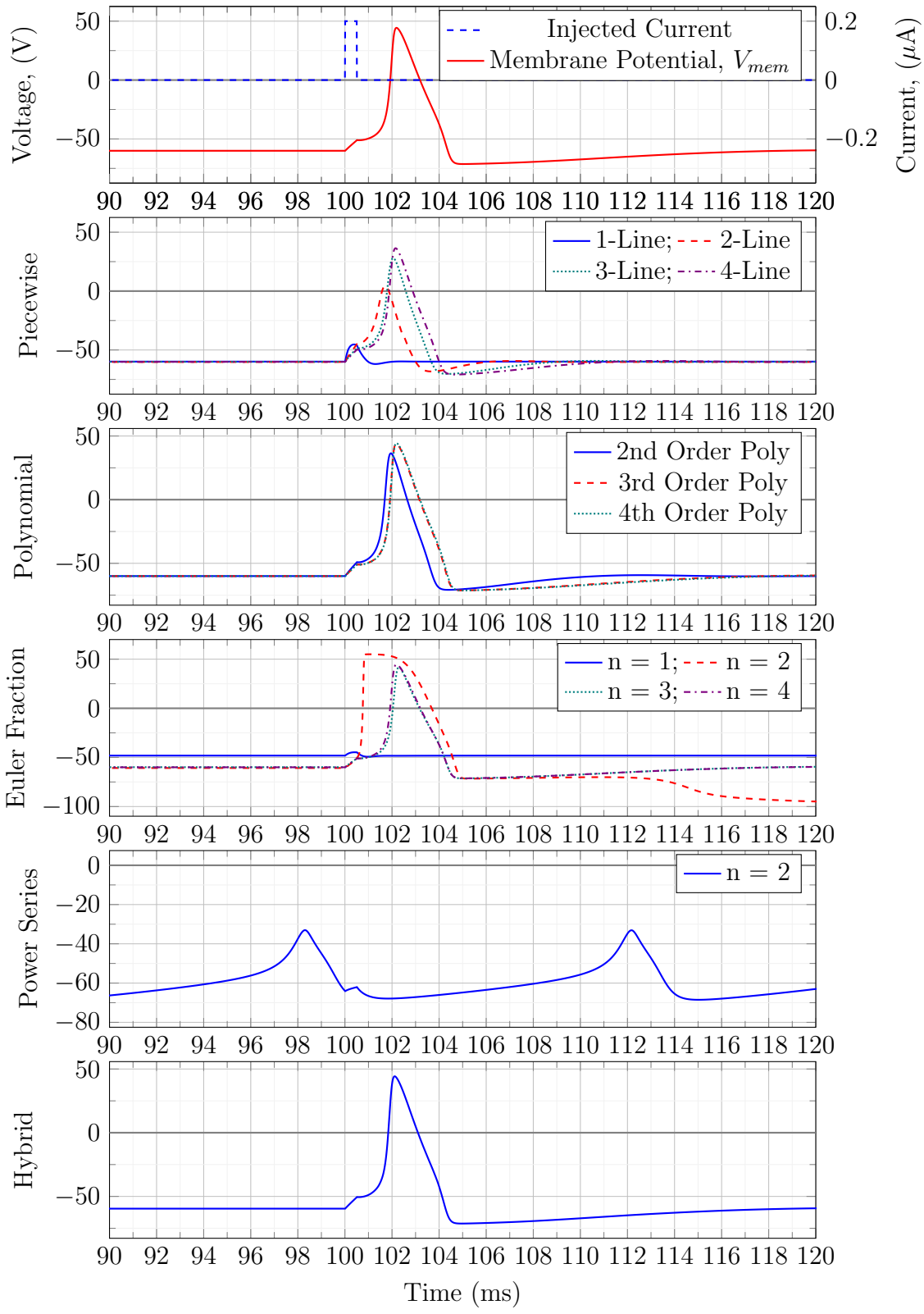


Figure 6.12: Comparative plots of the membrane potential, V_{mem}, for a Hodgkin Huxley neuron utilising each of the approximation models. A 0.5ms input stimulus of $0.2\mu A$ was used to trigger the AP.

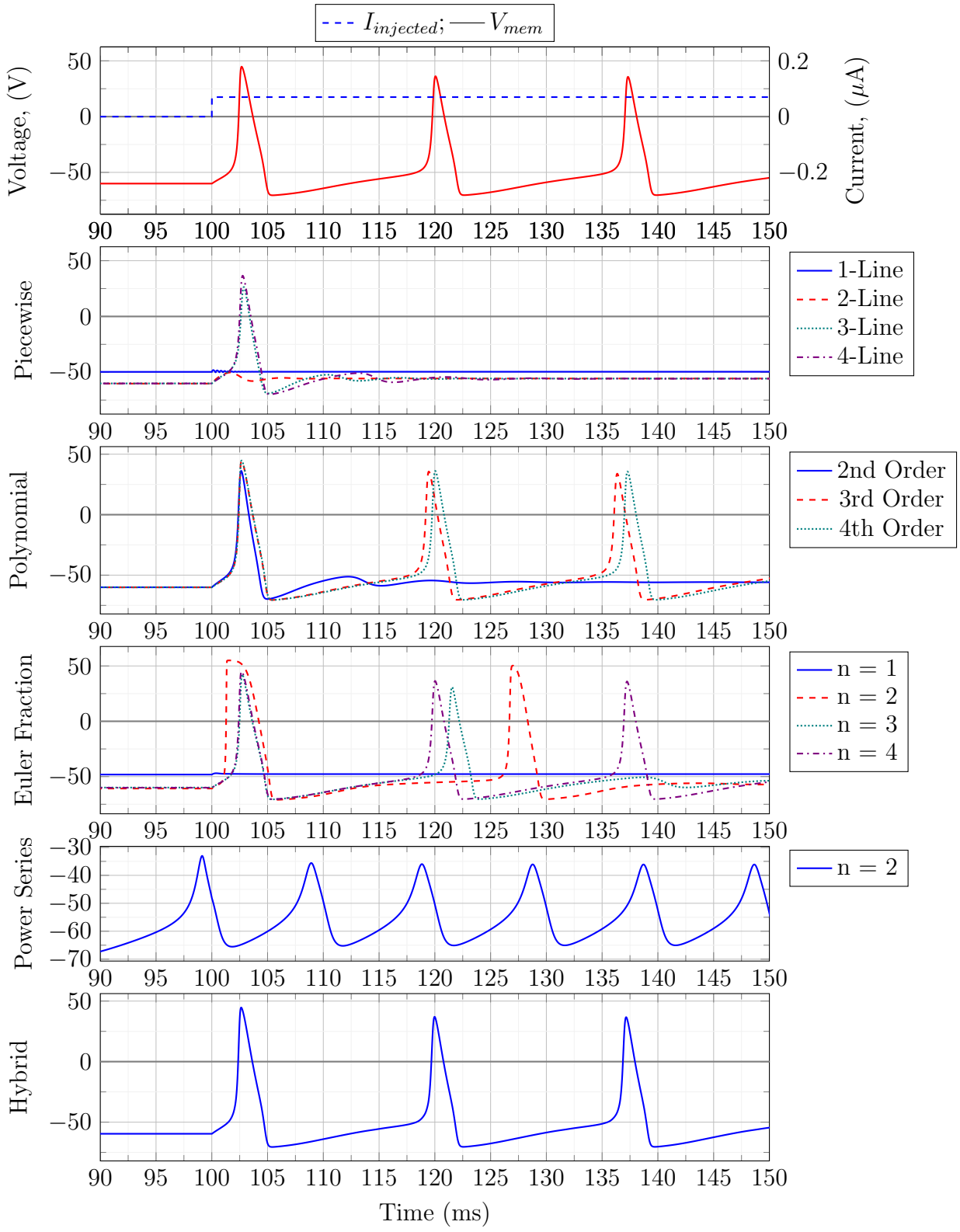


Figure 6.13: Comparative plots of the membrane potential, V_{mem}, for a Hodgkin Huxley neuron utilising each of the approximation models. A near threshold $0.07\mu A$ step input stimulus was used to trigger the neural spiking behaviour.

Table 6.3: Approximation performance for three different fundamental neuron functions. In DC resting '✓' represents less than 0.1% error, '≈' represents less than 1% error and '−' represents ≥ 1% error. Phasic spiking is judged according to the models ability to: ✓) generate an AP with correct timing and recovery windows; ≈) generate a characteristic AP in response to the stimulus. With tonic spiking, the models are judged according to their ability to: ✓) generate a pulse train of APs of the correct frequency; ≈) generate a pulse train of characteristic APs that is triggered by the input stimulus.

Approximation	DC Resting	Phasic Spiking	Tonic Spiking
1-Line Piecewise	≈	−	−
2-Line Piecewise	≈	−	−
3-Line Piecewise	✓	≈	−
4-Line Piecewise	✓	≈	−
5-Line Piecewise	✓	≈	−
6-Line Piecewise	✓	≈	≈
7-Line Piecewise	✓	≈	−
8-Line Piecewise	✓	≈	≈
2nd Order Polynomial	✓	≈	−
3rd Order Polynomial	✓	✓	≈
4th Order Polynomial	✓	✓	✓
Euler Fraction, n = 1	−	−	−
Euler Fraction, n = 2	−	−	−
Euler Fraction, n = 3	≈	≈	−
Euler Fraction, n = 4	✓	✓	✓
Power Series, n = 1	−	−	−
Power Series, n = 2	−	−	−
Power Series, n = 3	−	−	−
Power Series, n = 4	≈	−	−
Power Series, n = 5	−	−	−
Power Series, n = 6	✓	−	−
Power Series, n = 17	✓	≈	−
Power Series, n = 18	✓	−	−
Power Series, n = 19	✓	≈	−
Power Series, n = 20	✓	≈	−
Power Series, n = 21	✓	≈	≈
Hybrid	≈	✓	✓

error. This theory is supported by the performance of the hybrid model that manages to achieve approximations of all three modes despite the relatively simple underlying approximative model.

The error in the DC resting potential for the hybrid model was 0.74% and is caused by the 1-line piecewise element of the model. This error can therefore be removed by instead using the 2nd order polynomial as the latter part of the hybrid model. Such a hybrid/polynomial model would provide a good approximation of all three neuron modes of operation.

It is clear that there is no absolute best model to select when choosing an exponential approximation. The effects of error on the final solution can be somewhat unpredictable, meaning that any approximative model should be used carefully with related findings checked against full models prior to publication. It has been shown that approximative models can produce very close representations of key neural function. Such approximative models will allow researchers to rapidly iterate on, and study, large scale arrangements of neurons; performing aggregate tests that are typically unreasonable with full biophysically accurate neural models.

6.4 Implementing the Approximations in Hardware

The impact of using approximations on the Hodgkin Huxley model has been considered without any measure of the resource requirement for implementation. Having shown that the approximations are suitable, this Section considers the cost of implementing such systems in hardware. Each of the approximations were implemented on an Altera Stratix V FPGA using SystemVerilog.

Tables 6.5 and 6.6 show the resource requirements and fitting results when implementing the non-iterative models on the Stratix V FPGA, Tables 6.7 and 6.8 show the results for the iterative models. These models were implemented using double precision floating point maths blocks to ensure that the results were of comparable precision to that of a standard modern day processor. For comparison against each of these approximations, the performance and requirements for Altera's own floating point double precision exponential block or 'megafunction' are provided in Table 6.4. These megafunctions are performance-optimized by Altera for their own FPGA devices, providing a good comparative metric when considering the effectiveness of a new function implementation on an Altera FPGA.

These results show that the non-iterative approximation models require fewer Adaptive Look-Up Tables (ALUTs) than Altera's own solution, with the piecewise and most of the polynomial solutions requiring less than half that of the megafunction. All of the non-iterative models use fewer of the relatively large DSP blocks, meaning that these

Table 6.4: Resource usage or performance for Altera's own implementation of one IEEE floating point double precision exponential megafunction (ALTFP_EXP) [117].

Resource Type	Resource Usage
ALUTs	2905
Registers	2285
DSP Blocks	58
FMax (MHz)	205.32

implementations require significantly less floorspace. The iterative models, on the other hand are much larger than that of the Megafunctions and other approximative models. Each of the approximations also quote a much lower maximum clock frequency (FMax), meaning that calculations will take longer when using the approximations. Despite this apparent slowdown, the approximate models can still outperform the Megafunction implementation when computing the results of multiple neurons. Taking the 4-line piecewise pipelined model as an example, which uses less than 37% of the ALUTs, 21% of the DSPs and 19% of the registers required by the Megafunction, it is reasonable to expect to fit around three of the 4-line piecewise models into the space of a single ALTFP_EXP Megafunction. Since this implementation is pipelined, this means that the approximate model could output 3 results per clock cycle, yielding a theoretical calculation speed of 247.56Hz which is faster than the Megafunctions quoted speed of 205.32Hz.

6.4.1 Hodgkin-Huxley Implementations

With the approximations demonstrated in hardware, a full Hodgkin-Huxley model was constructed for the Stratix V FPGA. The top-level structure of this implementation is outlined in Figure 6.14, showing how the exponential operations were used alongside standard IEEE floating point operations to implement the Hodgkin-Huxley discrete mathematical model. The results for the 4-line piecewise approximation based implementation are shown in Figure 6.15. In this result, and each of the other functional approximation models identified in Table 6.3, the characteristic AP shape may be seen clearly at the location of the pulse stimulus.

The underlying maths blocks utilised in this study support half-, single-, and double-precision. Thus it was possible to rapidly test the implementation on a number of different precisions. In all cases, however, it was found that the Hodgkin-Huxley model would fail unless implemented with double-precision. For this reason, the half- and single-precision resource usage and speeds are not quoted within this thesis.

Table 6.5: Resource requirements for the approximate mathematical expansions of e^x. Implementations are using double precision floating point accuracy.

Approximation	ALMs Needed [=A-B+C]	ALMs used in Final Placement [A]	Estimate of ALMs Recoverable by Dense Packing [B]	Estimate of ALMs Unavailable [C]
1-Line Piecewise	617.0 (617.0)	621.5 (621.5)	4.5 (4.5)	0.0 (0.0)
1-Line Piecewise (pipelined)	617.0 (617.0)	676.0 (676.0)	59.0 (59.0)	0.0 (0.0)
2-Line Piecewise	659.0 (659.0)	673.0 (673.0)	14.0 (14.0)	0.0 (0.0)
2-Line Piecewise (pipelined)	670.0 (670.0)	712.5 (712.5)	43.0 (43.0)	0.5 (0.5)
4-Line Piecewise	691.0 (691.0)	705.5 (705.5)	14.5 (14.5)	0.0 (0.0)
4-Line Piecewise (pipelined)	681.5 (681.5)	734.0 (734.0)	53.0 (53.0)	0.5 (0.5)
2nd Order Polynomial	742.5 (742.5)	755.0 (755.0)	14.0 (14.0)	1.5 (1.5)
2nd Order Polynomial (pipelined)	831.5 (831.5)	898.0 (898.0)	71.5 (71.5)	5.0 (5.0)
3rd Order Polynomial	792.5 (792.5)	813.0 (813.0)	22.0 (22.0)	1.5 (1.5)
3rd Order Polynomial (pipelined)	1174.5 (1174.5)	1264.5 (1264.5)	103.0 (103.0)	13.0 (13.0)
Hybrid/1-Line Piecewise	1606.5 (364.0)	2169.0 (390.4)	570.5 (29.6)	8.0 (3.1)
Hybrid/1-Line Piecewise (pipelined)	1597.0 (371.6)	2140.0 (429.2)	552.0 (60.8)	9.0 (3.3)
Hybrid/2-Line Piecewise (pipelined)	1647.0 (377.8)	2189.5 (429.8)	552.0 (55.7)	9.5 (3.7)
Hybrid/4-Line Piecewise	1664.0 (372.7)	2207.5 (394.2)	549.0 (25.0)	5.5 (3.6)
Hybrid/4-Line Piecewise (pipelined)	1653.0 (368.2)	2176.5 (418.3)	528.5 (53.2)	5.0 (3.1)
Hybrid/2nd Order Polynomial	1746.5 (371.2)	2271.5 (395.8)	535.0 (27.8)	10.0 (3.1)
Hybrid/2nd Order Polynomial (pipelined)	1816.0 (370.3)	2302.5 (386.2)	503.5 (20.2)	17.0 (4.3)
Hybrid/3rd Order Polynomial	1798.0 (376.9)	2381.5 (399.5)	597.0 (28.3)	13.5 (5.7)
Hybrid/3rd Order Polynomial (pipelined)	2131.0 (378.7)	2742.5 (396.1)	622.5 (20.9)	11.0 (3.5)

Table 6.6: Resource requirements and timing analysis for the approximate mathematical expansions of e^x. Implementations are using double precision floating point accuracy.

Approximation	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted FMax (MHz)	0°C Restricted FMax (MHz)
1-Line Piecewise	968 (968)	197 (197)	8	85.19	82.44
1-Line Piecewise (pipelined)	959 (959)	389 (389)	8	86.75	83.58
2-Line Piecewise	1034 (1034)	198 (198)	12	65.02	64.91
2-Line Piecewise (pipelined)	1051 (1051)	397 (397)	12	87.95	83.99
4-Line Piecewise	1076 (1076)	198 (198)	12	66.52	65.84
4-Line Piecewise (pipelined)	1070 (1070)	421 (421)	12	85.93	82.52
2nd Order Polynomial	1218 (1218)	197 (197)	20	42.91	41.84
2nd Order Polynomial (pipelined)	1385 (1385)	505 (505)	23	81.81	81.47
3rd Order Polynomial	1317 (1317)	197 (197)	28	29.79	29.06
3rd Order Polynomial (pipelined)	2035 (2035)	747 (747)	37	80.85	81.79
Hybrid/1-Line Piecewise	2118 (504)	2864 (131)	8	87.44	84.38
Hybrid/1-Line Piecewise (pipelined)	2096 (507)	2864 (195)	8	82.47	79.7
Hybrid/2-Line Piecewise (pipelined)	2201 (506)	2872 (195)	12	83.96	81.06
Hybrid/4-Line Piecewise	2211 (510)	2869 (131)	12	62.74	62.3
Hybrid/4-Line Piecewise (pipelined)	2220 (506)	2896 (195)	12	86.52	83.05
Hybrid/2nd Order Polynomial	2356 (505)	2864 (131)	20	41.1	40.27
Hybrid/2nd Order Polynomial (pipelined)	2515 (505)	2916 (131)	23	83.31	82.13
Hybrid/3rd Order Polynomial	2454 (505)	2864 (131)	28	29.64	29.05
Hybrid/3rd Order Polynomial (pipelined)	3186 (505)	3288 (131)	37	80.11	79.96

Table 6.7: Resource requirements for Euler and power series mathematical expansions of e^x. Implementations are using double precision floating point accuracy. In each case the model has been unrolled to provide optimal speed performance. In the case of rolled iterative models the 1 iteration size may be used as there is minimal overhead required for such looping operations.

Approximation	Iterations	ALMs Needed [=A-B+C]	ALMs used in Final Placement [A]	Estimate of ALMs Recoverable by Dense Packing [B]	Estimate of ALMs Unavailable [C]
Euler Continued Fraction	1	13799.5 (1313.4)	20799.0 (1424.9)	7062.5 (111.7)	63.0 (0.3)
Euler Continued Fraction	2	20734.6 (1845.7)	31373.1 (1981.0)	10764.0 (135.9)	125.5 (0.6)
Euler Continued Fraction	3	27344.5 (1930.0)	31808.0 (2078.0)	4566.0 (148.3)	102.5 (0.3)
Euler Continued Fraction	4	34218.5 (2335.0)	38249.9 (2460.5)	4176.0 (125.9)	144.5 (0.5)
Power Series	1	6502.5 (686.2)	9698.0 (748.0)	3222.5 (61.8)	27.0 (0.1)
Power Series	2	12219.0 (1166.9)	18351.5 (1266.2)	6201.0 (99.4)	68.5 (0.2)
Power Series	3	18076.0 (1698.2)	26978.0 (1834.0)	9021.0 (136.0)	119.0 (0.3)
Power Series	4	23842.0 (2191.3)	35512.5 (2324.5)	11849.0 (133.6)	178.5 (0.4)
Power Series	5	29265.0 (2217.7)	33969.5 (2401.7)	4869.5 (184.6)	165.0 (0.6)
Power Series	6	35123.5 (2660.6)	38743.0 (2763.6)	3782.5 (104.4)	163.0 (1.3)

Table 6.8: Resource requirements and timing analysis for Euler and power series mathematical expansions of e^x. Implementations are using double precision floating point accuracy. In each case the model has been unrolled to provide optimal speed performance. In the case of rolled iterative models the 1 iteration size may be used as there is minimal overhead required for such looping operations.

Approximation	Iterations	Combinational ALUTs	Dedicated Logic Registers	DSP Blocks	85°C Restricted FMax (MHz)	0°C Restricted FMax (MHz)
Euler Continued Fraction	1	12342 (2071)	37192 (0)	32	98.26	94.67
Euler Continued Fraction	2	17022 (2861)	57769 (0)	44	97.1	93.05
Euler Continued Fraction	3	21818 (3622)	78378 (0)	56	92.81	89.76
Euler Continued Fraction	4	26455 (4412)	98987 (0)	68	93.08	89.61
Power Series	1	6215 (1042)	16892 (65)	16	99.46	96.4
Power Series	2	10981 (1817)	32963 (65)	32	94.53	90.48
Power Series	3	15819 (2609)	49071 (65)	48	95.54	93.08
Power Series	4	20459 (3392)	65177 (65)	64	91.7	90.23
Power Series	5	25452 (4205)	81283 (65)	80	95.21	91.73
Power Series	6	30404 (4976)	97391 (65)	96	93.72	91.91

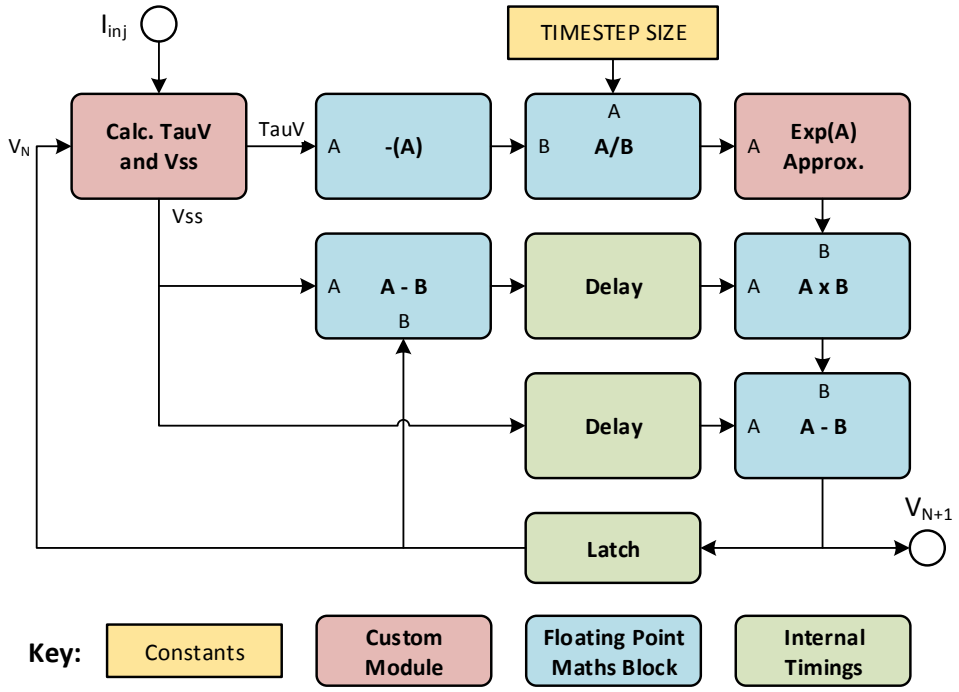


Figure 6.14: Top-level arrangement for the Hodgkin-Huxley FPGA implementation, showing how the exponential approximations were used alongside full IEEE floating point maths operations.

6.5 Conclusions

Many different approximations to the Hodgkin-Huxley model have been proposed and implemented within this chapter. It has been shown that high accuracy about the origin is required for effective tonic spiking replication, while good accuracy across the full supported range is also needed to reliably produce the characteristic AP morphology. The linear piecewise models are therefore not suitable for neuron modelling unless a large number of linear segments are used. In this case, the error about the origin is the constraining factor, meaning that a functional piecewise model (such as the 8-line model) provides far greater accuracy across the full range of approximated values than is actually required.

The polynomial models provide a greater accuracy than their linear counterparts. In these models, the error about the origin is still likely the constraining factor. In Table 6.2 it may be seen that the 2nd order polynomial model produces comparable error to that of the optimised 3-line model. A direct implementation of the polynomial would require 2 addition and 3 multiply blocks, while the 3-line model requires 1 addition and 1 multiply block alongside logic to select and store the 6 weights used in the generation of each line. Whether the additional logic and registers represent a better investment of available resource depends largely on what technology the final solution

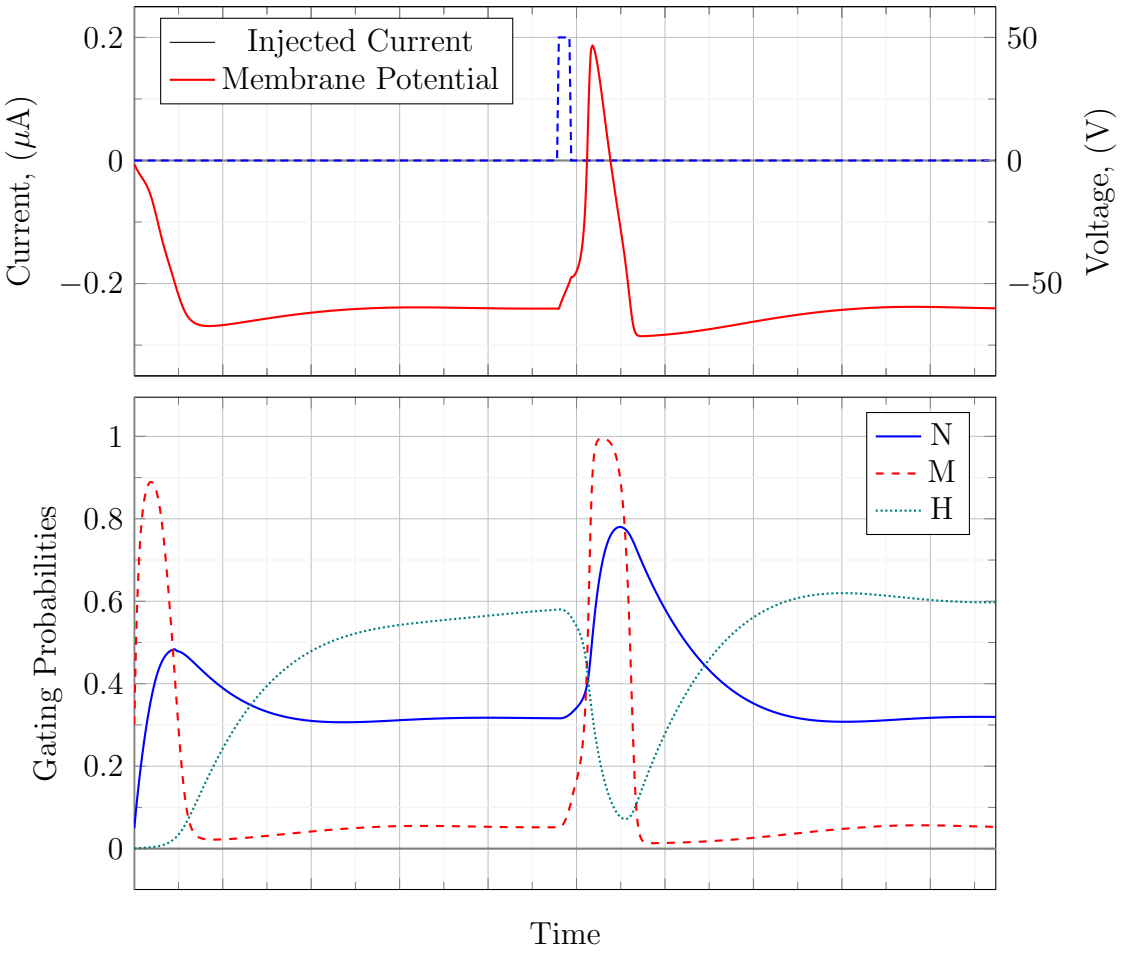


Figure 6.15: Hidgkin-Huxley model implemented on a Stratix V FPGA using a pipelined four line piecewise approximative exponential block. The characteristic membrane potential response to a $0.2\mu A$ pulse may be seen alongside the internal gating probabilities responsible for such action.

is implemented upon and it is therefore impossible to state categorically which model is better. With both the piecewise and polynomial methods it must also be remembered that additional logic is required to compute the integer and fractional parts of the floating point number. This additional logic is included in the quoted hardware sizes of Tables 6.5 to 6.8.

The Euler's continued fraction model required a large number of resources in implementation, but the iterative nature of this model provides designers with a variable precision solution. Matching the full range performance of the base-2 models (such as the piecewise and polynomial models) is impossible as it would require the iteration count to tend to ∞ to provide the required accuracy at extreme input values.

The power series produced very poor representations of the Hodgkin-Huxley model, requiring large numbers of iterations before even a single AP was generated. This is largely due to the fact that the Hodgkin-Huxley model operates within the negative domain for a lot of its internal variables. While the power series is a direct representation of the exponential function, it is not accurate for negative values and therefore not suitable for this application.

The hybrid model represents a unique combination of two different approximations, enabling the designer to select the accuracy for target regions. In the default hybrid model, a single line was used to approximate all values above $x = 0.0802$. This line was selected to use only a single addition operation, ensuring that the hardware requirement is minimal. For smaller values, the small value approximation $e^x = 1 + x$ was used. Despite its simplicity, this model achieved full representation of the Hodgkin-Huxley model, with only a slight deviance in the resting potential. This deviance can be removed fully using a more complicated large input approximation, such as the 3-line piecewise or 2nd order polynomial approximations. This model benefits from having minimal error for small values while also providing constant maximum error for any range selected.

This work has shown that approximations may be used to provide good and functional digital hardware Hodgkin-Huxley implementations. It was found that very-high accuracy about the origin is required to generate a functionally equivalent tonic spiking response. Additional, the accuracy across the entire range influences the models ability to generate the characteristic APs. This full-range accuracy need not be as strict as the small-signal accuracy, meaning models may utilise more approximative methods for large input signals without suffering failures in representation. The hybrid model uses 10 fewer multiply operations and 1 less addition operation when compared against the 4th order polynomial model, the only other non-iterative model to achieve accurate phasic and tonic spiking responses. These seemingly small improvements stand to have significant compound effects on the overall system when many thousands of neurons are being simulated. In the recent work of the Human Brain Project, for example, a simulation of macaque visual cortical areas was demonstrated - using

197,936 neurons within the primary visual cortex [118]. In such a project the hybrid model would provide a reduction of 1,979,360 multiply operations and 197,936 addition operations in each time-step, greatly reducing the computational time and power required. The application of suitable approximative methods is therefore a crucial element in the continued advancement and scaling of biophysically accurate neuron simulations. In cases where biophysical accuracy is not required, however, computationally efficient models may be used to further improve on the computational costs associated with the simulation. Such models may still benefit from optimisation and approximation techniques. Chapter 7 considers these models, showing how further acceleration may be achieved in such applications.

Chapter 7

Hardware Accelerated Activation Functions

A number of effective approximations for the exponential function, implemented as part of a discrete time Hodgkin-Huxley model have been discussed in Chapter 6. These approximation were shown to produce functionally equivalent neuron performance, comparing the approximated models response to both step and pulse stimuli against the original Hodgkin-Huxley model. Such computationally intensive implementations are necessary when biophysical accuracy is required, however there are many cases where this requirement does not apply. When this requirement is removed, artificial neuron models that offer enhanced efficiency and performance are more desirable. A number of these models are identified and described in Section 3.2. Typically, they use activation functions to provide their non-linear operation, as discussed in Section 3.3. While these models are more easily implemented than their biophysically accurate counterparts, they still require the application of resource intensive exponential and reciprocal operations to compute the results of their internal non-linear activation functions. Specifically, the popular *Sigmoid* function requires both an exponential and reciprocal operation. The cost associated with these operations scales rapidly in large neural networks of hundreds of thousands to millions of neurons. The power consumption and speed of individual neuron models have therefore become critical figures of merit as Artificial Neural Networks (ANNs) have grown in scale, where small improvements to the individual neurons leads to compound improvements for the whole system [33].

Building upon the work of Chapter 6, this chapter considers the acceleration and implementation of activation functions for large-scale ANN implementations. Section 7.1 reviews the methods currently used to accelerate digital activation functions. The common activation functions used in ANNs are then restated in Section 7.2, before introducing a number of custom sigmoidal-shaped activation functions, along with hardware focused approximative models in Section 7.3. Validation is a critical step when developing any new model. The datasets used to validate the models are introduced

in Section 7.4 before the simplest approximative model is shown to yield a resulting network accuracy within ±1% of an identically sized logistic sigmoid network when tested against the MNIST dataset in Section 7.5. Finally a hardware version of the model is introduced in Section 7.6, with the findings discussed in Section 7.7.

7.1 Accelerating Artificial Neurons

Repetitive and computationally intensive tasks can often be efficiently accelerated by adding additional specialist hardware. As an example, it is common to utilise a Graphical Processing Unit (GPU), shown in Figure 7.1b, to perform largely parallel mathematical calculations. In this case, the specialist design of the GPU offers improved performance over the more general Central Processing Unit (CPU), shown in Figure 7.1a. Custom hardware can also be added to a processors silicon to accelerate a specific operation, known as a *custom instruction* and demonstrated in Figure 7.1c. Custom instructions can yield even greater acceleration if the task is well defined and unlikely to change. Just as a CPU has a predefined set of specific instructions it can perform, a custom instruction allows the processor to utilise the custom hardware, performing one specific task with high efficiency or speed. Hardware acceleration is a critical factor in accelerating both neural network training, and inference, as they continue to grow in complexity and scale.

Beyond custom instructions, fully custom systems may also be applied to efficiently accelerate and implement ANNs. Unlike custom instructions, which merely add to a pre-defined CPU architecture, fully custom systems are designed from scratch to perform a given task with high efficiency or speed. This means that these systems often have considerable differences from standard computational architectures. Many such stand-alone hardware systems currently support the application of massive-scale ANNs, such as SpiNNaker [91], Neurogrid [98] and TrueNorth [94], that aim to provide large scales whilst maintaining energy efficiency through custom architectures and flexible hardware implementations. GPUs, Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) have also been used to provide state-of-the-art acceleration for machine learning. Of these three, GPUs often provide the most significant speed improvements at the expense of higher power requirements. This is largely due to the considerable investment that has been made into the production of fast and efficient high-power GPU products. Their availability and general purpose application within computing systems makes them a convenient solution for machine learning. ASICs can produce significantly better energy efficiency at moderate speeds by virtue of being designed explicitly for machine learning, however the long development cycle of approximately 6 to 12 months and prohibitively high fabrication costs often means that such solutions are not suitable for research. Commercial solutions, such as GraphCore do exist, however these are rare and their development requires considerable expertise and investment. FPGAs provide a flexible and cost effective method for specifying custom hardware, yielding speed and energy improvements over general

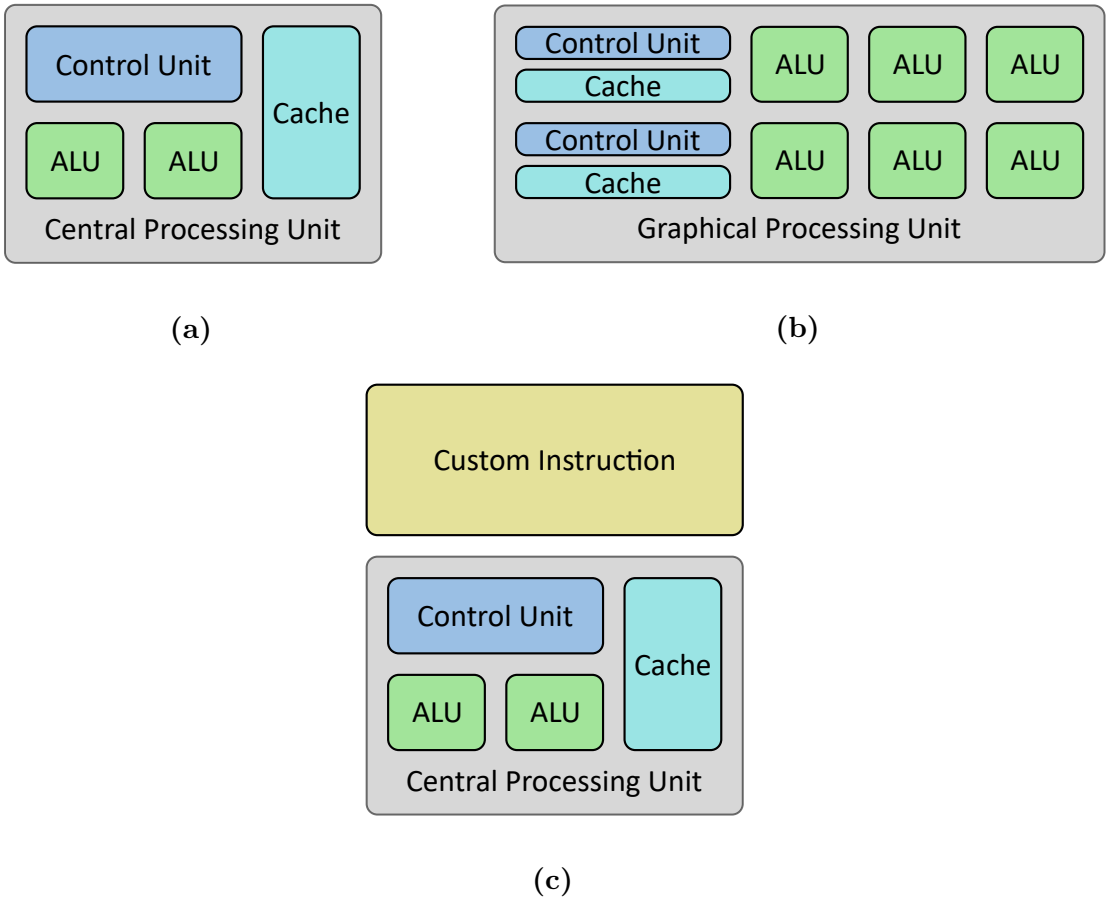


Figure 7.1: CPUs and GPUs leverage different architectures to achieve optimal performance when performing different tasks. The CPU, shown in **(a)**, has a single large cache, control unit and fewer Arithmetic Logic Units (ALUs) than the GPU, shown in **(b)**. Additionally, hardware may be added to implement custom instructions alongside standard processors, as shown in **(c)**.

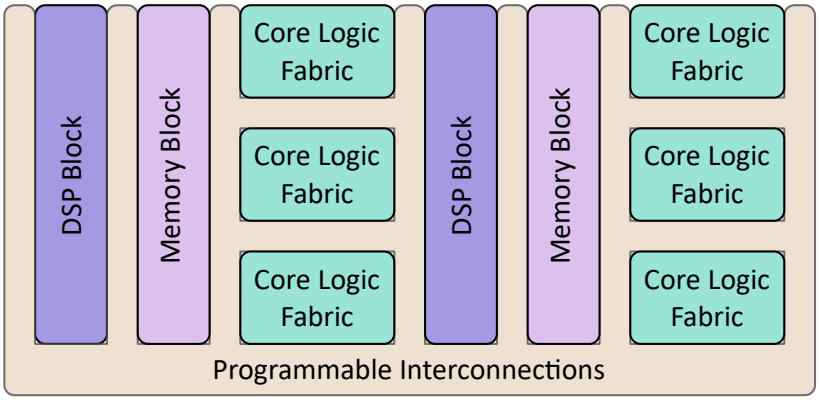


Figure 7.2: Example FPGA architecture, showing the customisable interconnection and core logic fabric that allows designers to specify and implement their own custom digital circuits.

purpose processors. Shown in Figure 7.2, FPGAs use a collection of programmable logic blocks, Digital Signal Processing (DSP) blocks and memory blocks, along with with programmable interconnect fabric allowing designers to specify and implement custom digital circuits. This flexibility, however, comes at a cost, with an FPGAs solution typically using slightly more power than a custom ASIC solution. This difference in power consumption is largely due to the additional routing circuitry that gives FPGAs their flexibility.

Each of these acceleration approaches typically attempts to approximate a pre-defined model or function as closely as possible, while optimising it for the platform in use. However, it is possible to specify functions that can be efficiently implemented in hardware with little-to-no compromise on computational performance. A faster and more efficient neural implementation may therefore be developed by instead taking the hardware constraints into consideration and designing a totally new model. Such models may be fully implemented in hardware, maintaining their 1st order continuity, or they may be further approximated to yield an even more efficient solution. Additionally, they may be specified in single or double precision making them suitable for direct implementation as a hardware accelerated custom instruction to provide speed improvements for processor implemented ANNs.

7.2 Activation Functions

In ANNs, each neuron commonly includes an activation function that is used to shape the output at each simulated time-step. Taking the general perceptron form introduced

Table 7.1: Common non-linear activation functions and their derivatives

Name	Activation Function	Derivative
Logistic Sigmoid Function	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)\left[1 - f(x)\right]$
Hyperbolic Tangent Function	$g(x) = \frac{2}{1+e^{-2x}} - 1$	$g'(x) = 1 - g(x)^2$
Gaussian Function	$h(x) = e^{-x^2}$	$h'(x) = -2xh(x)$
Rectifier (ReLU) Function	$y(x) = max(0, x)$	$f'(x) = (x > 0)$
Softplus Function	$z(x) = ln(1 + e^x)$	$z'(x) = \frac{1}{1+e^{-x}}$

in Section 3.2.2, this may be mathematically described as follows:

$$Y = F\left(\sum_{i=0}^{i=n} (W_i X_i) + B\right) \tag{7.1}$$

where Y is the output, W_i and X_i are the i-th weight and input respectively, n is the total number of inputs (both excitatory and inhibitory), B is the bias and $F(x)$ is the activation function. The activation functions used are typically non-linear as this provides the computational potential when the neurons are arranged into structured layers [35]. From Equation 7.1 it is clear that these non-linear activation functions can easily represent the most computationally intensive task, with the remainder of the model using addition and multiplication operations. It is therefore logical to start with the activation function to see whether it may be improved through implementation as a custom instruction. A set of common activation functions, first introduced in Section 3.3, is restated along with their first order derivatives in Table 7.1.

When selecting or defining an activation function there are many criteria that must be considered. In a gradient descent trained system, these criteria include speed of calculation for both the function *and its derivative*. Frequently the function must also be asymptotic at the extremities and have 1st-order continuity to ensure a smooth descent of the problem space whilst training. Without this continuity, areas of the problem space will have an infinite gradient meaning that it is possible for the tuned parameter to become trapped in local minima or repeatedly oscillate about the global minimum. Two common activation functions used in ANNs are the logistic sigmoid and the Rectified Linear Unit (ReLU), the reasons for their popularity highlight some of the fundamental characteristics required when designing new activation functions.

7.2.1 Logistic Sigmoid

The logistic sigmoid function is historically the most commonly used activation function in ANNs. This function boasts an asymptotic, monotonic, non-linear output with a simple derivative form that may be expressed in terms of the function itself. Despite these advantages, the logistic sigmoid function includes both a natural exponential and reciprocal operation, making it computationally expensive for large scale networks. Often, these operations utilise iterative methods to calculate a result, making considerable use of add, bit-shift and multiply circuitry. Such an approach leads to large area consumption and internal delays when compared against simpler maths operations. Many approximations of sigmoid activation functions have been produced in order to provide smaller or faster implementations.

Kwan demonstrated a sigmoid-like second-order piecewise activation function that was successfully applied to multilayer Feed-Forward Neural Network (FFNN) training [119]. Later, Faiedh *et. al.* presented a polynomial approximation of both the sigmoid activation function and its derivative, using single-precision floating-point representation [120]. Basterretxea *et. al.* proposed a recursive piecewise linear approximation scheme, showing high approximation accuracy with low memory requirements [115]. Gupta *et. al.* suggested an analogue conductance based model that used transistor asymmetry in cross-couple differential pairs to generate the sigmoid and its derivative [121]. Namin *et. al.* chose to use linear piecewise approximation, along with a Look-Up Table (LUT), to emulate a hyperbolic tangent sigmoidal function [122]. Tsai *et. al.* demonstrated a hardware-focused sigmoid function calculator, and Gomar *et. al.* produced a similar hardware model that achieved a 99.97% similarity to that of the original sigmoid function [123, 114]. As shown, there are many published approximations, however these often result in discontinuity in the first derivative. This loss of continuity has adverse effects on a gradient descent training algorithm.

7.2.2 ReLU

Within the last few years, ReLU has become one of the most common activation functions within ANN research, with considerable application in deep learning problems. Its success is founded in its simplicity of implementation, requiring a mere sign check to select both the output and derivative values. Despite this, there are cases where ReLU is not suitable (for example in control systems) due to a lack of continuity in the differential, unbounded outputs, and non-differentiable zero conditions. It has also been shown that a suitably initialised sigmoidal network can outperform deep ReLU networks, with the formers capability to achieve isometry found to be impossible by the ReLU counterparts [47].

The *softplus* function may be used as an approximation of the ReLU function to provide differential continuity. This function makes use of exponential elements and has the

logistic sigmoid function as its first order derivative. It therefore stands to reason that any improvements in the sigmoid model may also be applied to the softplus function offering improvements during network training.

7.3 Hardware Optimised Functions

It is clear from Table 7.1 that activation functions and their derivatives commonly use division and exponential operations to achieve their non-linearities. These computationally expensive operations must be addressed if activation functions are to be improved in speed or efficiency. The remainder of this chapter introduces and validates a novel implementation of modified activation functions, providing accelerated ANN performance.

The natural exponentials found in many activation functions are often computed using piecewise approximation, LUTs, or analogue non-linear dynamics. Another approach, however, is to use the relationship shown in Equation 7.2.

$$e^x = 2^{[xln(2)]} \tag{7.2}$$

Since multiplying a binary number by 2 equates to a bit-shift in hardware, an integer power of two only requires simple logic. Additionally, since the input to each activation function is already scaled by input weights, the $ln(2)$ scale factor seen in Equation 7.2 may be removed by instead scaling the stored weights within the network. This reduces the need to calculate this multiplication on each neuron output, modifying Equation 7.1 as follows:

$$Y = F\left(\sum_{i=0}^{i=n}([W_i ln(2)]X_i) + Bln(2)\right) \tag{7.3}$$

Using this method of weight scaling, the exponentials in each activation function may be replaced with 2^x. This removes the requirement for the calculation of a natural exponent, shown in Figure 6.4 to be $> 4\times$ slower than an Addition operation on a modern processor, replacing it with a significantly simpler bit-shift operation.

7.3.1 Redefined Logistic Sigmoid

It is important to consider the impact on the differential when modifying an activation function in this way. Indeed, the relative advantage of replacing the natural exponent with a power to the base-2 may be deemed worthless if it makes the differential

significantly more complicated to calculate. Applied to the logistic sigmoid function, as shown in Equation 7.4, the differential may be calculated as follows:

$$f(x) = \frac{1}{1+2^{-x}} = (1+2^{-x})^{-1} \tag{7.4}$$

Therefore:

$$f'(x) = \frac{d}{dx}(1+2^{-x})^{-1} \tag{7.5}$$

Applying the chain rule to Equation 7.5 yields equation 7.6.

$$\begin{aligned} f'(x) &= -(1+2^{-x})^{-2} \times \frac{d}{dx}(1+2^{-x}) \\ &= -(1+2^{-x})^{-2} \times -2^{-x} ln(2) \\ &= \frac{2^{-x} ln(2)}{(1+2^{-x})^2} \end{aligned} \tag{7.6}$$

This may be expanded and re-written as follows:

$$\begin{aligned} f'(x) &= \frac{1-1+2^{-x}}{(1+2^{-x})^2} \cdot ln(2) \\ &= \left[\frac{1+2^{-x}}{(1+2^{-x})^2} - \frac{1}{(1+2^{-x})^2}\right] \cdot ln(2) \\ &= \left[\frac{1}{1+2^{-x}} - \left(\frac{1}{1+2^{-x}}\right)^{-2}\right] \cdot ln(2) \end{aligned} \tag{7.7}$$

Substituting Equation 7.4 into Equation 7.7 yields:

$$f'(x) = \left[f(x) - f^2(x)\right] \cdot ln(2) \tag{7.8}$$

Finally this yields the same form as that of the original sigmoid equation, as shown in Equation 7.9.

$$f'(x) = f(x)\left[1 - f(x)\right] \cdot ln(2) \tag{7.9}$$

7.3.2 Redefined Activation Functions

The logistic sigmoid, TanH, Gaussian and Softplus functions may all be converted into a base-2 format using this method. In each case the original differential forms are maintained, as shown in Table 7.2. Each of the differential equations have an additional $ln(2)$ scaling factor, however this may be removed by instead incorporating it into the learning rates used during training. Many training algorithms utilise a scaling factor to control the rate at which the network is modified while undergoing training. Known as the learning rate, this value may be scaled to account for the additional $ln(2)$ in the differentials, removing the need for an extra multiply operation during the calculations.

Table 7.2: Proposed base-2 activation functions and their derivatives

Function Type	Activation Function	Derivative
Logistic Sigmoid	$f(x) = \frac{1}{1+2^{-x}}$	$f'(x) = ln(2) \cdot f(x)\,[1 - f(x)]$
Hyperbolic Tangent	$g(x) = \frac{2}{1+2^{-2x}} - 1$	$g'(x) = ln(2) \cdot [1 - g(x)^2]$
Gaussian	$h(x) = 2^{-x^2}$	$h'(x) = ln(2) \cdot [-2xh(x)]$
Softplus	$z(x) = ln(1 + 2^x)$	$z'(x) = ln(2) \cdot \frac{1}{1+2^{-x}}$

7.3.3 Converting Activation Functions Post Training

The base-2 activation functions shown in Table 7.2 are mathematically equivalent to their traditional exponential counterparts when the weights and learning rate have been scaled by a factor of $ln(2)$. This scaling factor means that it is possible to convert between the two activation function forms with relative ease. This enables designers to move between the two activation function models, meaning that networks trained with the traditional activation functions may be optimised post-training. To achieve this conversion the weights and learning rate must undergo the following scaling:

$$\begin{aligned} W_2 &= ln(2) \cdot W_e \\ l_2 &= ln(2) \cdot l_e \end{aligned} \tag{7.10}$$

where W_2 and W_e are the weights and l_2 and l_e are the learning rates for the base-2 model and exponential model respectively. Once this conversion has been performed the activation function may be swapped out without any re-training required.

Designers may also reverse this process, converting optimised models back into their full forms. This allows solutions to be moved freely between the optimised and full

implementations, greatly increasing the chances of adoption for any new optimised hardware model that is built upon the base-2 models. This one-to-one equivalence makes these new neuron models applicable in all cases where the original natural exponent models are used. The relative computational gains achieved through the application of the base-2 models is therefore available to both new and long-standing implementations, meaning an efficient implementation of the new models will have a significant impact on neural network solutions.

7.3.4 Decimal Power Approximations

Integer powers of base 2 maintain a simple hardware equivalence (as seen in Chapter 6), while the decimal powers add greater complexity. Unlike the biophysically accurate models discussed in Chapter 6, however, the artificial activation function approximations may diverge from the original model values so long as the performance of the network is not negatively impacted. The impact of any differences in absolute value will be negligible since ANNs use training algorithms to map the non-linear activation function onto the problem space. In this case it is the preservation of the underlying properties of the activation functions, such as the general shape and continuity of the equations, that is critical to ensuring that training will achieve a comparable mapping of the problem set.

By rephrasing the power to separate the integer and decimal values as shown in equation 7.11, it is possible to separate the implementation into two parts. The integer power, or bit-shift, and the decimal power, which will require approximation. This limits the approximation input to a range of 0 to 1, enabling a smaller implementation for a given accuracy when compared against a full approximation of 2^x.

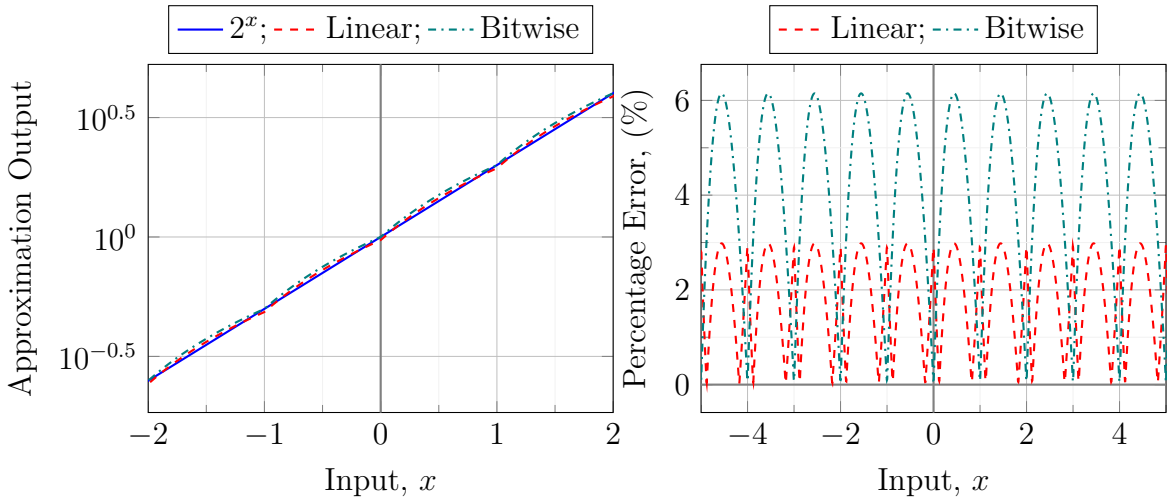
$$2^{i.d} = 2^i \times 2^{0.d} \tag{7.11}$$

The same approximation methods commonly applied to activation functions may also be used to approximate the decimal component shown in Equation 7.11. Figure 7.3a illustrates a 1-line linear piecewise approximation for 2^x, which uses Equation 7.12. The error for this approximation may be seen in Figure 7.3b to fall within ±3%.

$$2^{i.d} \approx 2^i \times (0.97018 \times d + 0.9702) \tag{7.12}$$

The approximation may be further simplified by recognising the similarity between the form of Equation 7.12 and that of an IEEE 754 double precision floating point number [124], shown in Equation 7.13.

$$Float\ Value = 2^{Exponent} \times (Raw\ Mantissa + Implicit\ One) \tag{7.13}$$



(a) 2^x Approximations **(b)** 2^x Approximation Error

Figure 7.3: The output (a) and percentage error (b) of 2^x, its 1-line approximation and the proposed bitwise approximation.

From this comparison it is clear that i may be treated as the exponent. Since $0.97 \approx 1$ the decimal power component could be approximated by simply using d as the raw mantissa. As d is already in the range 0 to 1 there is no scaling required to perform this operation. This bitwise replacement would require a single add to apply the exponent offset, however the value $i.d$ would first need to be represented in a fixed-bit format to allow isolation of the integer and decimal parts. Mathematically, this approximation may be represented as follows:

$$2^{i.d} \approx 2^i \times (d + 1.0) \tag{7.14}$$

where the input is identified as an integer part i and decimal part d. The output and percentage error associated with this technique may be seen in Figure 7.3. This novel approximative model stands to significantly reduce the computational cost of activation functions. It is important, however, to first assess whether these approximations and equations have any negative impact on neural network performance when compared against their original versions.

7.4 Datasets

Some benchmark must first be defined in order to measure any effect the new activation functions may have on ANN performance. The newly proposed activation functions must be tested against the original exponential versions to allow for a meaningful and fair comparison. For this validation, datasets must be chosen or generated, allowing the

performance of the different activation functions to be assessed in a controlled manner. In this work one established dataset and 7 generated datasets were used to test the activation functions.

It is the impact of the new activation functions on the network performance that will determine if the optimised activation functions are viable replacements for existing solutions. For this reason the activation functions must be tested in-network, using a consistent network topology and training algorithm. The training datasets must be delivered to the network in the same order to avoid any external effects on network training. With these precautions in place, any difference in post training network performance will be as a direct result of the activation function in use. It is therefore important to select a collection of different datasets, resulting in a distribution of tests that will reveal any general trends in activation function performance.

7.4.1 The MNIST Dataset

The Modified National Institute of Standards and Technology (MNIST) dataset is a large collection of labelled handwritten digits. This dataset is one of the most established benchmarks for image classification networks. The dataset was developed in response to concerns with existing handwriting recognition benchmarks, such as the issue of small or over-varied data sets. Originally selected by Chris Burges and Corinna Cortes from a collection of US National Institute of Standards and Technology (NIST) handwriting databases, MNIST is split into two distinct parts, a training set of 60,000 fully labelled examples, and a test set of 10,000 fully labelled examples. A modified version generated by Yann LeCun has since seen considerable application within published works making it a strong benchmark to assess the capability of new systems [5, 125]. While the original NIST utilised bounding boxes to centre the data, the LeCun version is modified to provide centring by centre-of-mass with large data windows. The digits are size-normalised and centred in a fixed-scale greyscale image. Following this pre-processing it is expected that humans could accurately and correctly classify the characters to within a 0.2% error [126]. A handful of random examples taken from the training set and paired with their associated labels are shown in Figure 7.4.

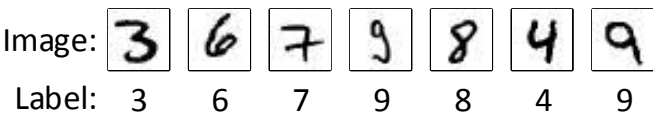


Figure 7.4: Example images and the associated labels taken from the MNIST training set.

Since its development, many network architectures and training algorithms have been tested against the MNIST dataset, providing a clear way to compare different approaches and solutions. In 2012, Ciresan *et al.* [127] almost matched human performance using a

Table 7.3: The main Scikit-Learn "make_classification()" parameters used in generating the classification datasets.

Parameter	Description
n_samples	The number of samples or data points.
n_features	The number of features or input values.
n_informative	The number of features that inform the correct class.
n_classes	The number of classes in the dataset.
n_clusters_per_class	The number of clusters used in generating each class.

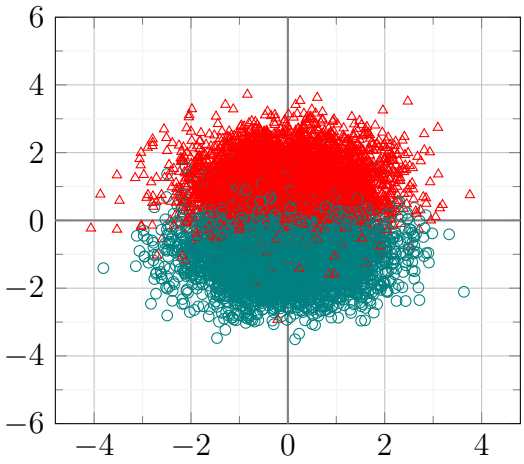
new multi-column Deep Neural Network (DNN) that boasts a performance of only 0.23% error. This same network was tested against other image classification benchmarks improving on state-of-the-art solutions in a number of them. A year later, Wan *et. al* demonstrated an ANN capable of 0.21% error on the MNIST dataset [128]. This network used Dropout, meaning that some weights were set to zero during training. Benchmarks like MNIST are critical in the comparison and assessment of ANN capabilities, showing that the gap between human and machine performance is closing rapidly in specific applications.

7.4.2 Generated Datasets

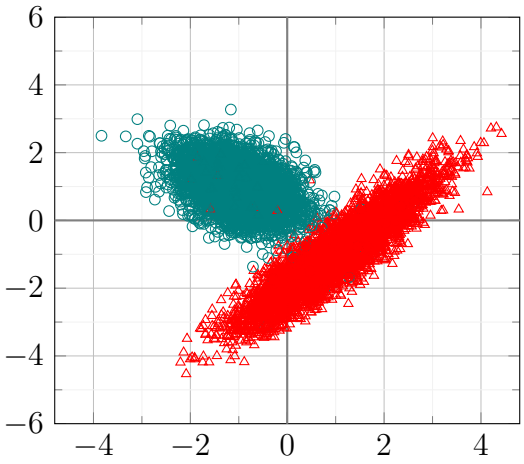
Whilst MNIST provides a realistic and practical benchmark for novel ANN implementations, it only tests networks against a specific problem-set or task. Networks that perform well using MNIST may fail in other applications. It is therefore critical to test new ANNs against a broad spectrum of problem class to ensure that any change in performance is detected. Generated datasets provide a means of testing a networks capability in mapping data with specific user-defined properties. These properties can be chosen to highlight specific features or operations. The volume of data that may be generated using these techniques makes this approach very useful when comparing different systems.

Generating this data by hand represents a significant time commitment, and it is therefore common practice to use programming techniques to procedurally generate the datasets. Python is a popular high-level programming language, commonly used in the machine learning community. The Scikit-Learn library is a machine learning library written in Python that includes a set of instructions for generating classification datasets with user-defined properties [129]. The command *sklearn.datasets.make_classification()*, for example, generates a random n-class classification problem according to its parameters, listed in Table 7.3. This Scikit-Learn library was used to generate the 7 additional datasets for comparing the activation functions.

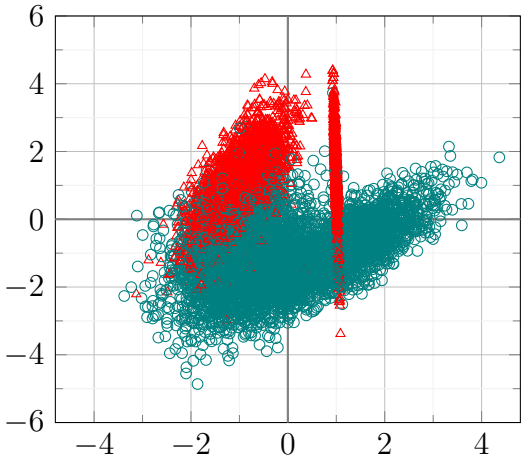
Four distinct classification datasets were generated using the *make_classification()*



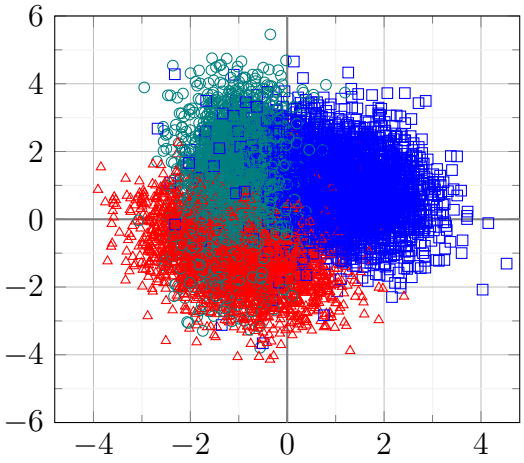
(a) Classification set A, where n_samples = 10000, n_features = 2, n_informative = 1 and n_clusters_per_class = 1



(b) Classification set B, where n_samples = 10000, n_features = 2, n_informative = 2 and n_clusters_per_class = 1



(c) Classification set C, where n_samples = 10000, n_features = 2, n_informative = 2 and n_clusters_per_class = 2



(d) Classification set D, where n_samples = 10000, n_features = 2, n_informative = 2, n_clusters_per_class = 1 and n_classes = 3

Figure 7.5: Generated classification datasets, with their generation parameters listed for each instance.

function. These are shown in Figure 7.5. These datasets were designed as incrementally challenging tasks. Classification set A, shown in Figure 7.5a, represents a simple task where a linear decision boundary sits along the axis. The clusters are intentionally mixed at the boundary meaning that no solution should achieve 100% accuracy. Without this uncertainty it is possible for networks to easily achieve perfect results, minimising the information that may be gained from comparing different network performance.

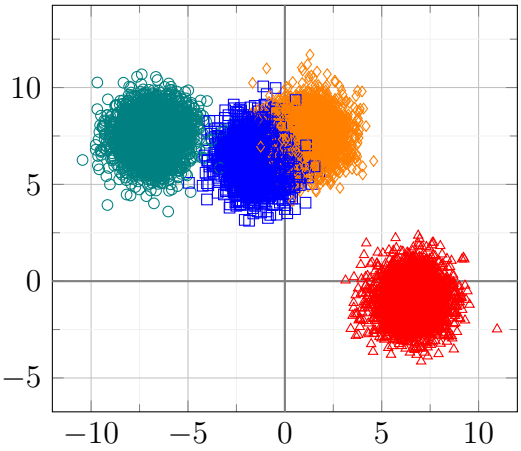
Classification set B, shown in Figure 7.5b, contains data defined by two inputs. Once again the clusters are bordering one another to ensure that the accuracy is not locked at 100% for any network. Classification set C also uses two input features. Unlike the previous sets, however, this data contains two clusters within each class. The secondary clusters from each class overlap one another within the same problem space, making this a challenging dataset to classify. This was generated to inspect how the new activation functions perform when dealing with unclear classification problems, as is often the case when working with real data. Finally, classification set D is shown in Figure 7.5d, and is a three class classification task where all classes closely border one another.

These classification datasets represent relatively simple linearly separable tasks (with the noted exception of Set C). Real data, however, is not always structured in this manner. Sometimes the classes are arranged in distinct regions, or 'blobs', in the problem space, as reflected in the dataset shown in Figure 7.6a. This data was generated using Scikit-Learns *make_blobs()* function, providing an additional benchmark for comparing the activation functions performance.

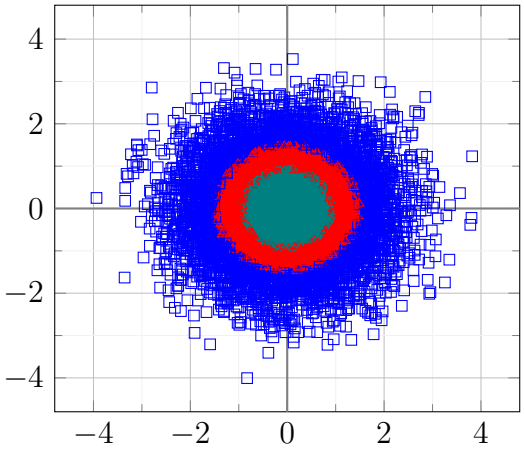
A Gaussian distribution problem was also designed, as shown in Figure 7.6b. In this dataset the classes sit within one another, with the inner classes entirely surrounded by the outer ones. The encapsulation of one dataset within another presents a unique classification challenge, requiring the system to correctly map the enclosed classes as closed regions within the problem space. Finally, a dataset comprising of two interlaced crescent shaped classes was generated, shown in Figure 7.6c. Each of these extended classification datasets represent types of problems seen within real data, making them useful in comparing the operation of different activation functions.

The datasets generated all contain two features, meaning that the data points sit within a two dimensional space. This was a design decision made to ensure that the ANNs problem space mapping could be visualised using a simple graph. The values at each point in the graph are generated by passing a comprehensive grid of input values into the networks. In practice many more inputs are used for classification problems, however the additional inputs merely result in increased network dimensionality and these smaller datasets are therefore still valid for comparing activation function performance.

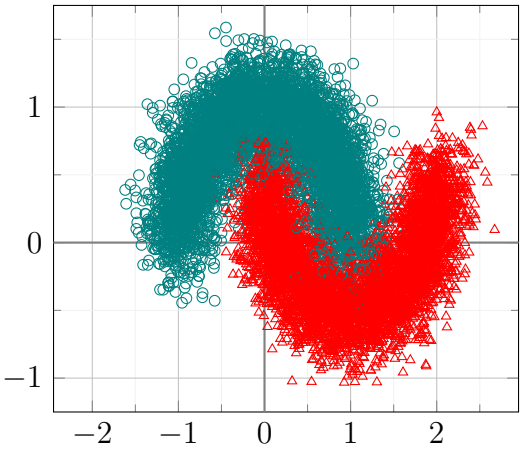
Each of these datasets was divided into 9000 training points and 1000 test points. These numbers were chosen to ensure that there were significantly more training points than test points and that the resulting performance measurements would be provided in steps of 0.1%. By dividing the data in this way the trained networks are more likely



(a) Blobs classification set, where n_samples = 10000, n_features = 2 and centers = 4.



(b) Gaussian classification set, where n_samples = 10000, n_features = 2 and n_classes = 3.



(c) Moons classification set, where n_samples = 10000 and noise = 0.2

Figure 7.6: Extended classification datasets, with their generation parameters listed for each instance.

to learn the underlying patterns of the data rather than just regurgitate the raw data itself.

7.5 Activation Function Performance Comparison

The newly defined datasets were used to test the activation functions, producing a comparative metric for each model. This allowed the classification accuracy of models using these activation functions to be compared against one another directly. The results from this comparison are presented later in Table 7.4. The activation functions chosen for these tests are restated in Equations 7.15 and 7.16.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{7.15}$$

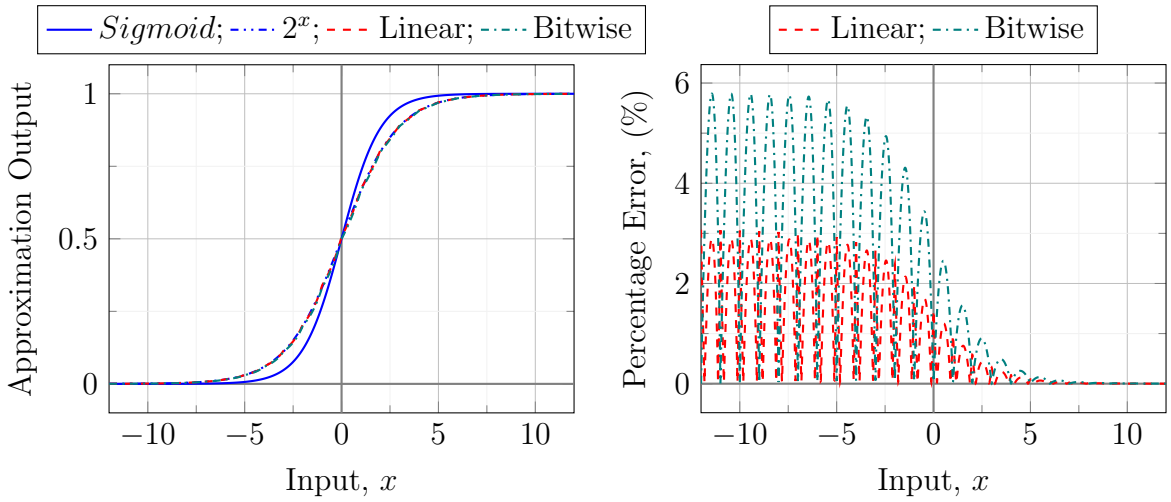
$$f(x) = \frac{1}{1 + 2^{-x}} \tag{7.16}$$

The linear approximation and bitwise approximation model discussed in Section 7.3.4 were also used alongside the full implementation models. These are defined below in Equations 7.17 and 7.18.

$$f(x) = \frac{1}{1 + 2^{-\lfloor x \rfloor} \cdot (0.97018\,(x - \lfloor x \rfloor) + 0.97018)} \tag{7.17}$$

$$f(x) = \frac{1}{1 + 2^{-\lfloor x \rfloor} \cdot ((x - \lfloor x \rfloor) + 1.0)} \tag{7.18}$$

Figure 7.7 shows each of the activation functions and their associated error. From these plots it is clear that each activation function maintains the sigmoid-like shape. The error for both approximations is largest for negative values, with large positive values becoming asymptomatically more accurate. The large negative value error makes sense when the output of the sigmoid function is considered. Since negative values produce small output values, only a small error is required to produce larger percentage error.



(a) Sigmoid Approximations **(b)** Sigmoid Approximation Error

Figure 7.7: The output (a) and percentage error (b) of 2^x, its 1-line approximation and the proposed bitwise approximation sigmoid functions.

7.5.1 Methodology

A neural network framework had to be developed to allow each of the chosen activation functions to be implemented and tested in a consistent and reliable manner. A neural network class was written in Python to support custom neural network structures, a code listing is provided in Appendix D. This class was written to utilise external activation functions, defined at runtime using the class parameters. Isolating the network class from the activation function definitions in this way allows new activation functions to be tested without considerable rework of the underlying test bench. The network class includes a simple back-propagation training regime to train the networks against a dataset.

The activation functions themselves were implemented as an additional Python library written in the C programming language, a code listing is provided in Appendix E. The use of the C programming language allows these functions to operate at a much lower computational level than implementations written in Python. At this low level, the floating point bit-wise elements within the function can be directly modified using similar techniques to that of equivalent hardware implementations. The binary approximation could therefore be tested without the considerable overhead otherwise required for bit manipulation through numerical means.

Each neural network was trained using the datasets described in Section 7.4. The simple back-propagation training algorithm was used for all training procedures. In each case the networks were exposed to the training data before a measure of performance was taken using the test data. Since the trained performance of a given network is influenced by random factors, such as the starting conditions of the network, this process was

repeated on a large number of networks to allow statistical analysis and comparisons. For the MNIST dataset, 256 networks were trained for each activation function, while 7680 networks were trained against each generated dataset for each activation function. The classification accuracy and training profile for each network was recorded to allow direct training performance comparisons between activation functions.

The training data was also presented to each network in a fixed order to ensure that any influence caused by data permutation was consistent across each of the networks under inspection. The basic back-propagation algorithm was used throughout the experiment since this test was designed to assess the performance of the new activation functions and not intended as an in depth analysis of different training algorithms. This training algorithm was detailed in Section 3.5.3.

Training any large network is a time consuming task, and the act of training several hundred networks for each dataset with each activation function presents a significant computational challenge. The Balena High Performance Computing (HPC) cluster at the University of Bath was therefore utilised to ensure that the results could be generated within an acceptable time frame. This HPC facility has 3,480 general purpose Intel "IvyBridge" and "Skylake" compute cores, with 16 and 24 cores per node, respectively. There is over 23 TiB of distributed memory, with 4 or 8 GiB/core, as well as an additional 2 nodes that have 512 GiB each for large memory-intensive jobs. There are a range of NVidia P100 and K20x General Purpose GPUs and Intel Xeon Phis (5110P) co-processors. The system has 0.7PBs of BeeGFS high-performance parallel file system and is connected by low-latency Intel TrueScale Infiniband at 40Gb/sec. This resource is shared amongst its users through a SLURM scheduler to ensure that computing tasks are performed within an acceptable time frame. This parallel computer enabled 255 networks to be trained in parallel, with a single core acting as a central management unit to collect and store the results. This combined with the HPC queue system enabled the computationally intensive task of training and testing these networks to be fully automated, and results were generated in a fraction of the time required if performed on a single machine. Each trained network was locked to a single core to ensure that the HPC didn't optimise one activation function solution over the others. This means that the results were equivalent to running the experiment on 255 stand-alone computers and ensured that each implementation had the same processing resource available.

Finally, as a means of testing the conversion methods introduced in Section 7.3.3, a total of 510 networks were trained with the original exponential based sigmoid activation function. These networks were then converted into the three base-2 activation function models, before testing them against the test data in each dataset. Any change in performance was monitored closely to inspect the effects of conversion or approximation post training.

Table 7.4: Mean dataset performance of the networks using each of the sigmoidal functions and approximations. The best performance for each of the datasets is marked with a * symbol.

		Mean Accuracy (%)			
Dataset	Network Size	Sigmoid	Base-2 Model	Linear Piecewise	Bitwise
MNIST	784-100-10	94.35 *	93.66	93.67	93.67
Blobs	2-20-5-4	89.41 *	84.57	84.70	84.78
Gaussian	2-50-5-3	90.17 *	88.47	88.47	88.47
Moons	2-20-2	85.72	86.62 *	86.59	86.58
Generated A	2-4-2	90.02	90.04 *	90.04 *	90.04 *
Generated B	2-4-2	99.30	99.30	99.30	99.30
Generated C	2-4-2	85.16	85.94 *	85.91	85.92
Generated D	2-20-5-3	80.22 *	80.18	80.16	80.16

7.5.2 Results

The logistic sigmoid, base-2 sigmoid, 1-line piecewise and bitwise activation functions were tested against each of the datasets, using a fixed number of training epochs for each network. In total, 256 networks were used for each activation function on the MNIST dataset while 7680 networks were used for each activation function on each of the generated datasets. The mean performance for the activation functions was then calculated and is shown in Table 7.4.

Following this initial test, the performance of the 510 converted networks for each activation function was also recorded. For each network, the original sigmoid activation function performance was subtracted from the results to yield a measure of the relative performance change as a direct outcome of post-training conversion. Figure 7.8 illustrates the results for this experiment. From this figure it is clear that the base-2 and sigmoid activation function models are functionally equivalent when conversion is applied, as there is no change in performance. The additional approximation used in the linear and bitwise activation functions results in a small spread of network performance. Since the approximations affect each and every node within the network, the overall effect is seen to produce a normal distribution within the performance change plots.

The results in Table 7.4 demonstrate that, while there is a change in classification accuracy when moving between the models, the difference between the full sigmoid and bitwise approximation model is relatively small. Within the converted results the spread in performance is seen to follow a slightly skewed normal distribution, with more than half the bitwise approximation networks falling within $\pm 0.015\%$ of the full model performance. Since the bitwise model was generated from the ground-up with a hardware implementation in mind, it is logical to now compare this model against the original sigmoid models computational efficiency and resource utilisation.

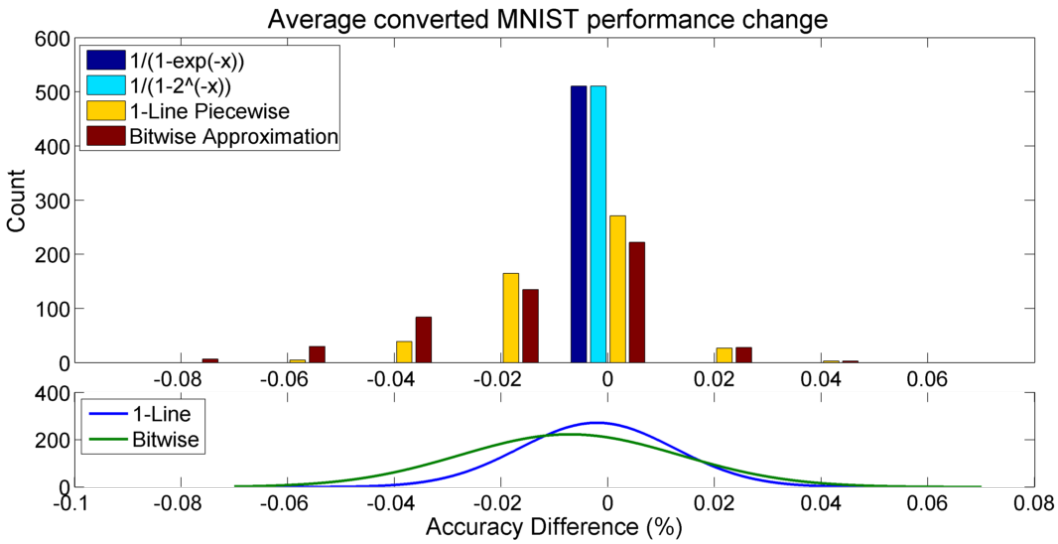


Figure 7.8: MNIST dataset performance difference in 510 converted networks, showing the change in the networks performance following a conversion from a logistic sigmoid function.

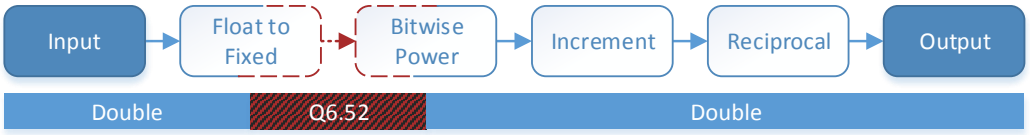


Figure 7.9: Flow diagram for hardware accelerated custom instruction, with data-type shown below.

7.6 Custom Instruction Implementation

The resource utilisation of the new bitwise model must be considered to ensure that it yields an efficient and effective neural network model before the relevance and meaning of the results in Section 7.5.2 may be assessed. A parametrised hardware version of the bitwise approximation model was developed to support double precision custom instruction acceleration. Figure 7.9 illustrates the process flow for this model, showing the fixed-width and double-precision elements. Since the bitwise approximation moves the fractional bits into the mantissa there is no benefit to having more fractional bits than the number of bits supported by the mantissa itself. Equally, the logistic sigmoid function is asymptotic and approaches 0 and 1 for values outside the range ± 15, therefore limiting the signed integer elements to 6 bits provides a suitable range so long as the conversion contains saturating logic. As such the fixed-width representation chosen for this double precision implementation is Q6.52.

This hardware system was synthesised on an Altera Cyclone SoC 5CSEMA5F31C6, an FPGA which includes an ARM A9 core. ARM is a well established processor manufacture, with their devices commonly found in low-power or mobile solutions. The

Table 7.5: Throughput for the sigmoid function on a number of processing architectures.

Hardware (or processor)	Core Clock Speed	Throughput	Clock Cycles per Calc.
Proposed Bitwise Accelerator	87.7 MHz	87.7 MHz	1
Intel Core i7-7800x	3.5 GHz	87.0 MHz	40
Intel Core i5-4460	3.2 GHz	71.4 MHz	45
Intel Xeon E5-2609	2.4 GHz	44.1 MHz	54
Intel Core i5-650	3.2 GHz	12.5 MHz	254
AMD 1090T	3.2 GHz	12.3 MHz	260

ARM A9 core is therefore a practical device to test custom instruction implementations. The final system yielded a requirement of 2604 Adaptive Logic Modules (ALMs), 4138 registers and 68 DSP blocks. This chipset was selected to allow easy integration of the custom instruction within the ARM core. The DSP block requirements are due to a 58-bit multiply performed in the Newton-Raphson Reciprocal block. The Altera tool suite includes a timing analysis tool called TimeQuest. Using TimeQuest it was shown that the proposed design could run at a maximum clock speed of 87.70 MHz. The system has an issue rate of 1 clock cycle, meaning that a valid output is available on each clock cycle. These features combined, the solution yields a custom instruction capable of 87.7 $MFlops$. To achieve this speed the system has been pipelined, meaning that a setup of 38 clock cycles is required before the first result becomes available. In large- to massive-scale networks there will be thousands of activation function calculations to perform meaning that such setup cycles are negligible when comparing the overall calculation times.

Using the synthesis tools power analyser (Quartus PowerPlay) the total consumption for this implementation is 76.59mW, comprising a dynamic power consumption of 43.71mW. The Newton-Raphson implementation used in this accelerator represents 87% of this power and uses most of the quoted DSP blocks. These values were found using the default '6_slow_1100mv_0c' model with a clock of 86.96MHz.

It is possible to compare the accelerated bitwise implementation to that of a sigmoid implementation on modern processors as shown in Table 7.5. The processor implementation used the standard C++ exponential function, which was called 1 billion times to calculate the average computation time. All values were double precision IEEE 754 floating point numbers to be consistent with the bitwise accelerated version. The CPU implementation ran on a single core for comparison against a single accelerator. In practice a custom-instruction accelerator could be implemented as either a single accelerator per CPU, as one accelerator per core or even as one accelerator per register. However, there is a trade-off between computation speed and size when deciding how to implement the custom instruction within the processor fabric itself.

While the proposed bitwise function throughput is comparable to that of the Intel Core

i7, this accelerated system has a clock speed of only 87.7 MHz and requires only 1 clock cycle per calculation leading to a considerably lower power consumption than any of the general purpose processors that have core speeds of at least 2.4 GHz.

7.7 Discussion

The results in Table 7.5 show that the proposed bitwise accelerator yields a comparable throughput to that of a modern high-end processor. Table 7.4 shows that the MNIST mean performance fell by less than 1% when using the bitwise implementation in place of a sigmoid activation function. At the same time, the number of clock cycles required per calculation was reduced by 97.5% when compared against an Intel Core i7-7800x. The throughput remains approximately constant despite the large differences in clock cycles. It is highly likely that variance seen in Table 7.4 for the generated datasets is due to the training profiles and the activation functions impact on training convergence. This theory is supported by the conversion results in Figure 7.8 which show that, while the approximations can cause some small changes in network performance, the full base-2 model can represent a sigmoid model perfectly with suitable scaling applied to the weights. Since the scale factor acts to influence the learning rate indirectly, it seems likely that the problem space is being scaled through use of these custom activation functions.

The hardware implementation is intended for use as a *custom-instruction* to accelerate a CPU or GPU using dedicated hardware. This concept is demonstrated in Figure 7.1c. Whilst a fully custom FPGA implementation would permit maximum performance it would be at the cost of accessibility and usability. Future processor technologies are increasingly likely to include FPGA fabric, providing users with a space for custom-instruction implementation on chip. This prediction is justified in the context of Intel's recent acquisition of Altera [130], alongside mention of Altera FPGA co-processors. Such FPGA-CPU pairings already exist and therefore make good targets for neural network implementations, however as this technology matures it is likely to become mainstream in commercial and industrial applications, making custom instruction design an important part of function acceleration.

7.8 Conclusions

Efficient and fast ANN implementations are required to support the continued advancement and scaling of neural network applications. The computational complexity of an activation function plays a considerable role in the computational cost of the network as a whole, with each neurons output requiring its own activation calculation. The sigmoid function represents a good approximation of learning rate seen in success-based learning within nature, however, in it's full state it cannot

compete with the computationally efficiency shown by the more abstract ReLU model. Unlike ReLU, the Sigmoid function has a continuous non-zero differential and asymptotic extremities. It also closely reflects a class probability distribution as shown in section 3.3.2. An efficient sigmoid implementation would allow large networks to leverage these properties without significantly compromising the computational efficiency of the system as a whole. Additionally, a continuous version of the ReLU function commonly used in gradient descent training, termed the *Softplus* function, has the Sigmoid equation as it's differential. Training this function therefore requires the calculation of the Sigmoid. For these reasons, a novel approach to implementing sigmoidal equations has been developed in this chapter, considering a direct solution and two approximative methods.

A conversion between the traditional sigmoid and the proposed base-2 sigmoid was also shown. This direct conversion mapping is crucial when designing new custom instructions and hardware accelerators, as it allows a network to be trained using the accelerated hardware before mapping it to a more standard sigmoid based model for use in regular computer architectures. While the whole hardware system was able to run at 87.70 MHz the accelerated bitwise power module actually had an upper speed of 443 MHz, making it significantly faster than the rest of the system. Unless the data already starts in fixed point representation, the requirement of an extra float-to-fixed converter must also be considered when looking at the performance for this bitwise power module. The float-to-fixed converter yielded a speed of 180 MHz, however this could be improved by building a pipelined custom addition block instead of using the standard addition operator within the module.

The approximations are suitable for use in double precision systems, yielding a good result when compared to the full sigmoid model. Both of these models were built using the assumption that the derivative would be approximately equal to that of the full base-2 model, however this in-built error did not appear to slow the learning rate of the system in any observable fashion. The robustness of these networks is surprising given that the bitwise approximation based activation function had a percentage error as large as 6% for large negative numbers. It should be noted that in this area the function output is rapidly approaching zero, and therefore even small changes to the output level can have significant impact on the percentage error of the function.

When compared with other general purpose processors, the hardware accelerator shows a significant speed improvement over older models and keeps pace with some of the latest CPU products. The number of clock cycles required to perform the sigmoid calculation has been reduced by a factor of 40 when compared with the next best processor. Given that this custom solution will be significantly more energy efficient than a general purpose processor, it is still beneficial to use such acceleration in large- to massive-scale networks even when the computation speeds are similar.

This chapter has focused upon accelerating the sigmoid function. However the methods introduced could easily be applied to the other base-2 activation functions shown in

Table 7.2. Additionally, this accelerated sigmoid function could be used for the softplus model to accelerate the derivative calculations and speed up training.

In the hardware implementation the accelerator provides an increase in the direct calculation speed of the sigmoid function - which in turn is measured as throughput. This makes the assumption that the memory bandwidth is high and the latency is low enough to ensure that the pipelined module is fully utilised. Given that the accelerator runs at 87 MHz this is not an unreasonable assumption. The surrounding delays and latency associated with other parts of a neural network implementation may become the dominant factor in the speed of a system using this accelerator, however there is considerable published work that clearly identifies a need for accelerated activation functions making this a relevant step in the acceleration of neural network implementations [114, 123, 122, 121, 115, 120, 119, 131]. Of course further optimisation could still be performed. Reworking the multiply within the reciprocal operation block, for example, and replacing it with an optimised pipelined version may offer improvements in maximum clock speeds or reductions in DSP block requirements

In this chapter a new base-2 activation function and a simple bitwise approximation for decimal powers of two that requires no mathematical operations was introduced. These proposed activation functions and approximation models have been tested against common and generated datasets to provide performance comparisons alongside the standard logistic sigmoid function. A double-precision floating point hardware accelerator was built using the bitwise approximation activation function allowing direct comparison with modern day processing solutions. To the authors knowledge this work represents the first double precision floating point sigmoid accelerator, capable of calculating a result to within 6 Unit in the Last Place (ULP). This hardware system was built in System Verilog and shown to be faster than modern day processors performing the same task. The clock cycles per calculation are compared against an Intel Core i7-7800x, with an improvement of 97.5% demonstrated when using the new activation function model.

Chapter 8

Utilising Neural Network Locality

The performance gains available from conventional processors are limited by fundamental constraints, as predicted by Moore's law and Dennard scaling. A paradigm change is required to develop more advanced and powerful processors that go beyond these observed trajectories. In many cases, nature can provide examples of powerful and intelligent processing approaches, from these it is possible to construct bio-inspired technologies, such as neuromorphic systems, which promise to address or even remove the current constraints. At present, however, there is a significant disparity between the performance of these systems in nature, and those implemented artificially. This chapter explores one of the fundamental differences between biological neural networks and Artificial Neural Networks (ANNs), that of inter-neuron connectivity. The connectivity seen in naturally occurring networks is very different to that of current hardware implementations, and designing neuromorphic systems based on the predominantly *local* connectivity seen in nature has the potential to dramatically improve performance and scaling in modern processing platforms. The movement of data, and the infrastructure to enable its communication, both use additional power and hardware resource. Processing solutions, such as the biological brain, that can both store and process data locally therefore stand to reduce this power and resource consumption. Additionally, the signal delay and signal attenuation seen in the biological brain is close to theoretical minimum values, while the number of synapses for a given volume is close to a theoretical maximum [132]. This suggests that the structure used within nature represents a heavily optimised system, and replication of its efficiency and density is therefore desirable.

As the scale of modern neural networks grows rapidly the interconnection requirements between neurons present an increasing challenge. Dedicated connection infrastructure, packet switched networks and cross-bar arrays have all been applied in attempt to deliver interconnectivity. These hardware implementations frequently assume exhaustive connectivity between neurons, with packet switched networks prone to packet loss and

non-deterministic signal delays, often resulting in bottlenecks when transmitting large quantities of data [133]. These issues are not found in biological neural networks. It should therefore be possible to develop new architectures to reduce the dependence on global communications by considering the connectivity of biological networks.

This chapter outlines existing reconfigurable architectures before introducing a novel locally-connected architecture for implementing biologically inspired neural networks in real-time. The proposed architecture is validated using the segmented locomotive model of the *C. Elegans*, performing a demonstration of forwards and backwards serpentine motion, as well as coiling behaviours. The concepts of locally and globally connected architectures are discussed, with local connectivity discovered to offer up to a 17.5× speed improvement over hybrid systems that use combinations of local and global infrastructure.

Section 8.1 presents an overview of connectivity in neural networks, highlighting a key difference between the connectivity seen in artificial and biological systems. Section 8.2 considers two architectures that 'close the gap', highlighting their differences and relative merits. One of these architectures, termed the grid architecture, is introduced in this section as a fully local communications system. Section 8.3 introduces the methodology used to validate this new novelly connected architecture, introducing both a 2D and 3D *C. Elegans* physical model that allows for direct visualization of movement as a result of neuron simulations. The results of these tests are presented in Section 8.4 and analysed in Section 8.5, where a speed improvement of up to 17.5× for a 50 segement *C.Elegans* locomotive model is shown when implemented on the fully local grid architecture. Finally, a discussion on the limitations of locally connected architectures is provided in Section 8.6, with the conclusions highlighted in Section 8.7.

The critical contributions of this work are the introduction of the novel grid architecture; the discovery that locally connected architectures can realise signifiant speed and scalability improvements over their hybrid-local and global counterparts; a demonstration that Convolutional Neural Networks (CNNs), which appear to contain a high degree of locality within their connectivity, map poorly to locally connected architectures; and the identification of differences in artificial and biological system dimensionality, revealing that both the connection infrastructure and underlying technologies require careful consideration when using biology as inspiration for new systems.

8.1 Neural Network Connectivity

The supported connectivity within any neuromorphic system forms an important metric in determining the functionality of the system as a whole. A system with too few connections cannot fully represent a large heavily-connected network, while a system

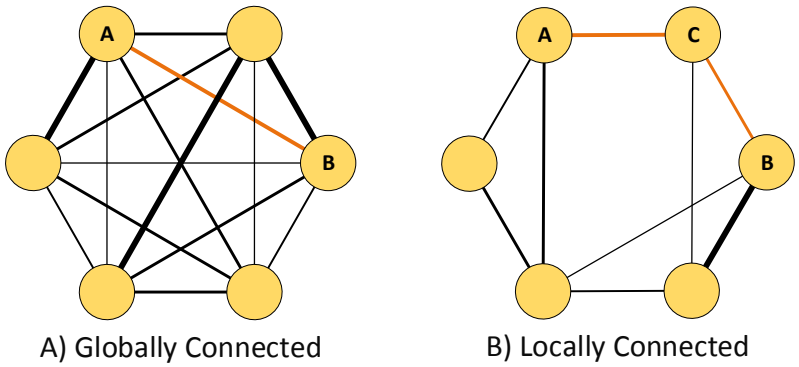


Figure 8.1: In globally connected networks every neuron is connected to every other neuron directly (or via some communications infrastructure). In locally connected networks this is not the case - instead there is a measure of locality to the interconnection meaning that direct signals cannot be passed between all neurons, as shown for neurons A and B where an additional neuron, C, sits along the communications path.

with too many connections wastes resource on redundant connection infrastructure. Finding a suitable balance for this neuron interconnectivity is therefore an important consideration when designing efficient and mobile neuromorphic systems. Since these systems typically operate as general purpose neuro-computing devices, it is common for such systems to use connection infrastructures that provide exhaustive neuron-to-neuron interconnection options resulting in a fully connected graph that may represent any network of suitable size. While this solution provides a high degree of flexibility when fitting new networks to hardware, it also often results in unused resource as few networks require this exhaustive one-to-one connection support.

In this work the term 'global connectivity' is used to refer to exhaustive solutions that allow any neuron to directly communicate with any other neuron within the network. It is important to note that this does not require unique dedicated hardware for each and every interconnection. Network based implementations are often used within this field to yield a high interconnectivity while sharing the resource through traditional network management techniques. The distinction between a 'globally connected' system and that of a 'locally connected' system is therefore found in a neurons ability to communicate with any other neuron within the system without requiring the assistance of any additional neurons acting as 'relays'. This concept is shown in Figure 8.1 where neurons A and B are connected directly in the global system, but not in the local system where an additional neuron, marked C, sits between the two neurons in question. In this way locally connected networks are sparser than globally connected systems.

8.1.1 Connectivity in Hardware

Despite the large variation in neuron models, the communication infrastructure used to link neurons together into large neural networks is often more consistent in design.

Global connectivity provides the greatest flexibility for reconfigurable architectures, and is therefore often assumed as a requirement for new neuromorphic architectures.

Packet-switched networks are the most common communication method used to pass signals from one neuron to another, with each packet typically containing the firing neurons identity and time of excitation. One such packet-switched network system is SpiNNaker [90, 91, 92], which was introduced in detail in Section 4.2. Each of the processor cores used within this system are surrounded by a light-weight packet-switched asynchronous communications infrastructure allowing any neuron within the system to communicate with any other neuron. This globally connected system is arranged into a 3D torus structure in an effort to reduce the maximal distance between any two points and therefore accelerate the long-range communications between neurons. One of the constraining factors of such designs is the communications channel, with implementations of highly active large-scale networks requiring significant network bandwidth. This bandwidth requirement creates a choke point in the system, slowing the speed at which simulations can be performed. In the worst case this can also lead to dropped packets within the communications infrastructure, meaning that signals and data can be lost during a simulation.

To avoid the issue of communications bottlenecks, unique connections in an all-to-all configuration can also be used. Such systems typically make use of cross-bar arrays and memristor structures to provide reconfigurable hardware interconnects in a small silicon package [134, 135]. These systems yield the fastest neuron to neuron throughput, however they scale at a rate of $O(N^2)$ making them unsuitable for large-scale networks of several thousand neurons. They will also contain significant redundant connectivity for any given application as many of the connections will go unused in a typical neural network.

Some hybrid systems, such as TrueNorth (introduced in Section 4.3) offer a mix of local and global connection infrastructure in attempts to reduce the unused connection count while limiting the long-distance communications bottleneck. These systems still provide global connectivity, however they use the local connection infrastructure to complement the global communications infrastructure and avoid it's over-utilisation. In TrueNorths case, (local) short-distance communications are handled by an on-chip 64K crossbar array and asynchronous routers, and (global) long-distance communications are handled by protocols for intra-chip connections [94]. TrueNorth also uses time-division multiplexing to allow a single computational unit to calculate the outputs of 256 logical neurons. These neuron outputs are then connected locally through an on-chip 64k synaptic crossbar array. The interchip connections are handled by a packet-switched network providing the full global connectivity.

Neurogrid (introduced in Section 4.4) is another hybrid system with a mixed analogue-digital implementation that operates in the deep sub-threshold region of the semiconductor devices. Synaptic inputs between spatially neighbouring neurons are distributed using a custom local arbor system, or a multi-cast router; while the global

core-to-core connections are handled using a binary tree routing network [98]. The local arbor system bears resemblance to general cross-bar array implementations, resulting in a fast and effective local connection infrastructure at the expense of high resource requirements.

A common denominator of neuromorphic platforms is the flexibility to handle arbitrary levels of connectivity between any and all neurons. Whilst this is a clear advantage in general solutions, for specific examples this can lead to loss of efficiency. Finding a balance between flexibility and reduced redundancy will play a critical role in the development of new and novel neuromorphic systems for low-power or mobile devices. One approach to manage this issue is the implementation of a neural fabric that can be optimally configured for different scales of connectivity, whilst retaining the reconfigurability of the large scale neuromorphic implementations.

8.1.2 Connectivity in Nature

Neuromorphic systems began with heavily bio-inspired solutions in an attempt to address the power consumption gap between biological systems and that of digital systems. This biological inspiration has continued to drive innovation and development within the field. The free living nematode *Caenorhabditis Elegans* (*C. Elegans*) has become a key component in scientific understanding of the wider function and arrangement of the biological nervous system. The *C. Elegans* is a transparent nematode that is approximately 1.3 mm long and takes 3.5 days to maturate. With only 302 neurons in its nervous system, the *C. Elegans* was the first organism to have its connectome (the neuronal 'wiring diagram') fully mapped [136]. The animal moves forwards and backwards in a serpentine fashion using rectilinear locomotion to locate food and escape predators. It may also exhibit coiling behaviour in the event of a sudden stimulus to the animals side. It has a muscular structure built up of 8 distinct regions that run down both sides of its body as shown in Figure 8.2. These muscles are used to produce locomotion through sinusoidal activation which propagates down the animal in a synchronised fashion.

The *C. Elegans* connectome is divided into a number of functional sub-systems, such as the locomotory system responsible for eliciting the animals characteristic motion. A model of this locomotory system (discussed later in Section 8.3.1), which contains a sub-set of 86 neurons and is connected by 180 synapses, was developed by Claverol *et. al.* [137] and Bailey [15] and demonstrated the neural patterns characteristic of forward, reverse and coiling motions in *C. Elegans*.

The natural division of the nervous system into smaller sections is also found in larger animals, such as rodents and small birds. This high locality in neural processing has allowed researchers to emulate complex systems, such as the auditory system of a bat, without needing to simulate the animals entire nervous system [138].

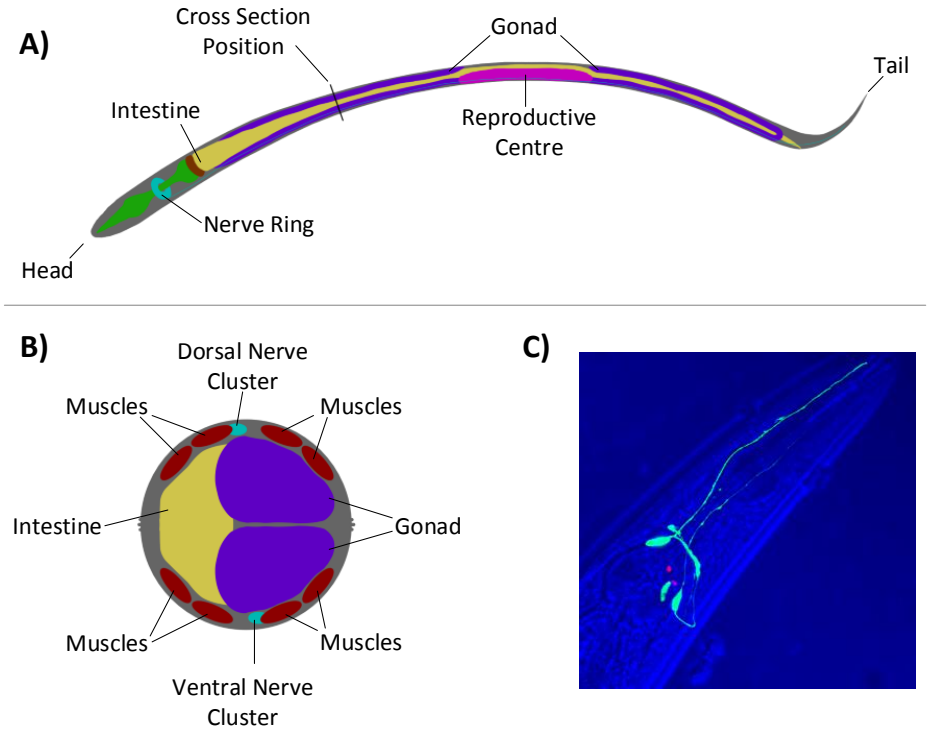


Figure 8.2: Anatomy of the *C. Elegans*: A) viewed laterally, showing the nerve ring located at the head end of the animal; B) Cross-section, showing the two major nerve clusters and 8 muscle structure that run down the animals length, adapted from images found at `www.wormatlas.org`; and C) A fluorescent subset of neurones within *C. Elegans* showing projections running alongside the pharynx. The overall structure of the head region can be seen in the blue, overlaid transmission. Image courtesy of John Chad and Ilya Zheludev.

The nervous systems of primates follows a similar functional hierarchy. However due to the increasing size and complexity, it can often be harder to explicitly identify neurons or regions responsible for specific actions. One example of a highly localised processing system in humans is the control mechanism behind the urinary bladder - there is a clear distinction between the roles of the somatic motor system and the sympathetic and parasympathetic nervous systems in the micturition of the bladder. The sympathetic innervation of the bladder originates in the lower thoracic and upper lumbar spinal cord segments where sympathetic activity causes the internal urethral sphincter to close. Activation of this pathway by a modest increase in bladder volume due to the accumulation of urine thus closes the internal sphincter and inhibits the contraction of the bladder. Urine is held in by the somatic motor innervation of the external urethral sphincter. Patients who have lost descending control of the sacral spinal cord continue to exhibit autonomic regulation of the bladder function, since micturition is stimulated reflexively at the level of the sacral cord by sufficient bladder distension, as measured by the stretch and pressure mechanoreceptor afferents within the pelvic sacral plexus [139]. Although the interactions are multiple, and complex, it is clear that bladder control largely occurs close to the bladder and the sacral roots, with only distant and occasional interaction from the encephalon.

Each sub-system is highly effective at specific tasks and demonstrates a largely localised solution which has a clear benefit in terms of energy cost and complexity. The level to which this specialisation and segmentation occurs in the Human Nervous System (HNS) remains an open question, however some small examples have already been identified and this is most evident in reflex systems found within the Peripheral Nervous System (PNS).

8.1.3 Measuring Connectivity

There is a clear difference between the connectivity seen within modern neuromorphic systems and biological nervous systems. This locality of connection, however, is hard to define in a quantitative manner as it is obscured by a networks layout and inherent complexity. A quantitative measure of locality would therefore be a useful metric when considering the differences between hardware and biological systems.

In the 1950s Rent discovered empirically that there was an inherent power-law relationship between the number of gates and the number of terminals within Very-Large Scale Integration (VLSI) circuits. This observation was later used to generate a scaling property known as "Rent's Rule" by Lanzerotti *et. al.* in 2005 [140], however the basis for this relationship was first explained earlier by Landman and Russo in 1971 [141]. This rule takes the general form:

$$T = kg^p \tag{8.1}$$

Table 8.1: Comparison of both physical and topological Rent coefficients, p, reproduced from *Communication Locality in Computation: Software, Chip Multiprocessors and Brains* by D. Greenfield [143].

Network	p (Physical)	p (Topological)
Globally Connected Architecture	1.0	1.0
VLSI	0.901 ± 0.006	0.73 ± 0.04
C. Elegans	0.740 ± 0.070	0.77 ± 0.06
Human MRI	0.828 ± 0.005	0.75 ± 0.07
Human DSI	0.782 ± 0.014	0.78 ± 0.07

where T is the number of terminals, g is the number of gates or internal components, p is the power-law exponent (between 0 and 1), and k is a constant of proportionality that was later interpreted by Christie and Stroobandt as the average number of terminals per gate [142]. Christie and Stroobandt derived Rent's rule theoretically for homogeneous systems, observing that a measure of the optimisation achieved in placement is reflected by the parameter p, known as the "Rent exponent". Low values of p represent systems with short interconnects while larger values represent systems with greater degrees of interconnection.

Rent's rule may be used to define the connectivity in hardware neural networks, relating the internal (local) and external (global) connections within the network [143]. In these networks, the terminals are synapses and the gates are neurons. An important aspect of Rent's rule is that if the equation is re-framed in terms of logarithms [144], the relationship between T and g becomes linear when both axes are expressed in logarithmic terms as shown in Eq. 8.2.

$$log(T) = log(k) + p * log(g) \tag{8.2}$$

This yields an offset value *log(k)* and a linear slope (in the log domain) determined by the Rent coefficient, p. One of the interesting observations in nature is that, as shown in Table 8.1, this coefficient tends to be the same for all neural structures, typically about ($p = 0.75$) for both lower order animals such as *C. Elegans*, and for human brains [144]. The globally connected neuromorphic systems, however, yield a Rent exponent of $p = 1$ due to their requirement for generalised solution representation. This difference in Rentian scaling suggests that some savings in resource may be achieved through the careful constraining of the interconnection infrastructure to explore the trade-off between local and global connectivity.

8.2 Utilising Locality in Neural Architectures

It has been shown that there is a difference between the interconnectivity of neuromorphic and biological networks. Section 8.2.1 details a hybrid-local system designed by Bailey [15] that uses locally connected synapses to alleviate the load on a global connection bus. A novel fully-local architecture is then introduced in section 8.2.2. Both of these systems have been designed as abstracted reconfigurable solutions, allowing individuals to implement networks using them without needing to edit or understand the underlying hardware infrastructure. This is similar to that of Field Programmable Gate Arrays (FPGAs), where an individual uses a high-level Hardware Description Language (HDL) to define a hardware-independent circuit which is then implemented using a hardware specific toolchain.

8.2.1 Hybrid Local - The Column Architecture

The *column architecture* is a neuromorphic system designed to leverage the locality of neuron connections seen in nature. This reconfigurable neural hardware system is capable of real-time simulation of neural networks, making use of hybrid-locality to reduce the communications bottlenecks. Designed using custom VHDL building blocks, the system may be programmed to simulate a desired network using a set of configuration words. In order to reduce the connectivity requirements, the neurons and synapses were developed as separate blocks and conceptually arranged into two columns, as shown in Figure 8.3. This allowed a novel local connection infrastructure to be used within the synapses, reducing the global bus communications requirements.

The neuron blocks were designed to operate in two modes, as either a base neuron or a pattern generator. In the base neuron mode the neuron takes input from the synapses, passes it through a threshold block and triggers a burst generator if the threshold is exceed. In the pattern generator mode, the input is replaced with an internal oscillator that triggers the burst generator at user defined intervals. These two modes may be programmed by the end user using the pre-defined configuration words, shown in Table 8.2. As such, this model allows the end user to customise the neurons contained within the network without requiring a detailed understanding of VHDL.

The synapse blocks perform the bulk communications within the system. Designed to support connections to the neighbouring synapses at both the input and output sides of the block, these synapses perform the summing operations for the various neuron stimuli while also providing a way for multiple synapses to be triggered concurrently by the same neuron. These local internal connections within the synapse column reduce the communications between the synapse and neuron columns considerably as each neuron need listen to only one synapse within the system. The various configurations these connections support are shown in Figure 8.4. Synapse 1 and synapse 2 are not connected in any way; synapse 3 has it's output connected to synapse 4, meaning that

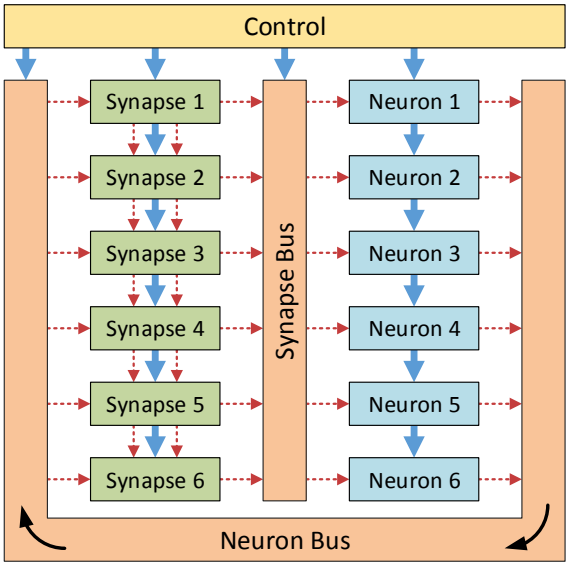


Figure 8.3: High level design of the column architecture. The two communications buses, the control module and connections (solid blue arrows) and the reconfigurable connections (dashed red arrows) are shown. Optional connections between the synapses allow for reductions in the bus utilisation - therefore reducing the risk of communications bottleneck issues.

Table 8.2: Configuration word for the neuron block operating in the base neuron mode.

Bits	Length	Function
7 - 0	8	Address
15 - 8	8	Excitation Threshold
23 - 16	8	Inhibition Threshold
32 - 24	8	Burst Length
47 - 32	16	Action Potential Time
63 - 48	16	Refractory Period

Table 8.3: Configuration word for the neuron block operating in the pattern generator mode.

Bits	Length	Function
7 - 0	8	Address
8	1	Enable Phase Offset
40 - 9	32	Period
72 - 41	32	Phase Offset
80 - 73	8	Burst Length
96 - 81	16	Action Potential Time
112 - 97	16	Refractory Period

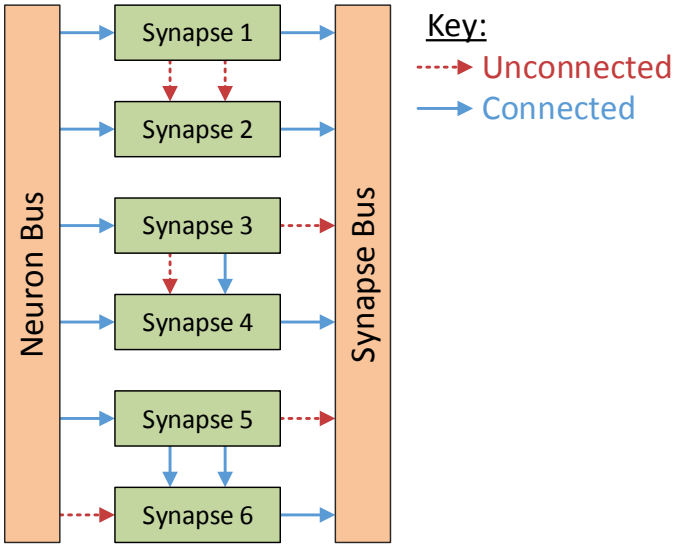


Figure 8.4: Synapse configuration examples. Synapse 1 and 2 are operating independently. The outputs of synapse 3 and synapse 4 are connected together, yielding the sum of these two blocks as an output. Synapse 5 and synapse 6 are connected together at both input and output to support concurrent activation of an already active synapse.

synapse 4 will output the sum of the two synapses; and synapse 5 is connected to synapse 6 at both the input and output stage. This last configuration allows a synapse to be activated by a pre-synaptic neuron even while the synapse is already active.

As with the neuron blocks, the synapse blocks are fully configured using the configuration word detailed in Table 8.4. This configuration is performed alongside the neuron configuration and once again allows full use of the system without prior knowledge of VHDL.

The neuron and synapse columns are connected together by two buses. These buses are global connections, allowing any neuron to connect to any synapse and vice versa. At any time only one neuron and one synapse may drive the neuron and synapse buses

Table 8.4: Configuration word for the synapse block.

Bits	Length	Function
7 - 0	8	Input Address
15 - 8	8	Output Address
23 - 16	8	Synaptic Weight
55 - 24	32	Synaptic Delay
87 - 56	32	Synaptic Duration
88	1	Input Link
89	1	Output Link

respectively, however many neurons and synapses may listen to these buses at the same time. To ensure that these buses are not multiply driven a global controller is used that cycles through the address space. The neuron (or synapse) that contains the active output address drives the neuron bus (or synapse bus) for that clock cycle, and any synapses which has the same address in their input address register will listen to the bus for that clock cycle. In this way the neuron and synapse bus use time-division multiplexing to appear fully transparent to the system, allowing full global connectivity. Since the system must cycle through all neurons in each simulated time step, the maximum system simulation clock frequency, F_c, is impacted by the number of neurons simulated, n, and may be found using Equation 8.3, where F_i is the internal system clock frequency.

$$F_c = \frac{F_i}{n} \tag{8.3}$$

This clock scaling creates significant problems when simulating large neural networks of several thousand neurons. Since each neuron must drive the bus once per simulated time step, the simulation clock must run much slower to accommodate for the increase in scale. This system is therefore not scalable. By ensuring that synapse to synapse connections were performed outside the scope of the wider connection infrastructure, however, this hybrid-local system achieved measured speed-ups of 20x that of previous models it was compared against [145].

8.2.2 Fully Local - The Grid Architecture

In order to overcome the fundamental scaling limitation of the column architecture, this work introduces a new grid based architecture that more closely reflects the locality seen within natural systems. Unlike the column architecture, this new grid architecture brings the neurons and synapses back together into a single block or node. These nodes are arranged into a 2D grid, connected in a North, East, South, West fashion to their direct neighbours using a custom connection infrastructure. The system is made up of two distinct elements, the node, which is an arbitrary neuron implementation and the IO block, operating as a connection to external actuators and sensors.

As shown in Figure 8.5, the connection infrastructure sits between each of the nodes and re-directs the incoming signals according to user-defined configuration bits. This allows the nodes to be connected into loops, with each neuron situated upon a total of four different loops - connected to one on each face. The loop data is internally latched within the nodes, allowing all nodes to drive their outgoing loop segment at all times.

A global controller cycles through the address space, 0 to m, where m is the node-count of the largest loop in the network. At address 0 all the nodes drive their outputs onto

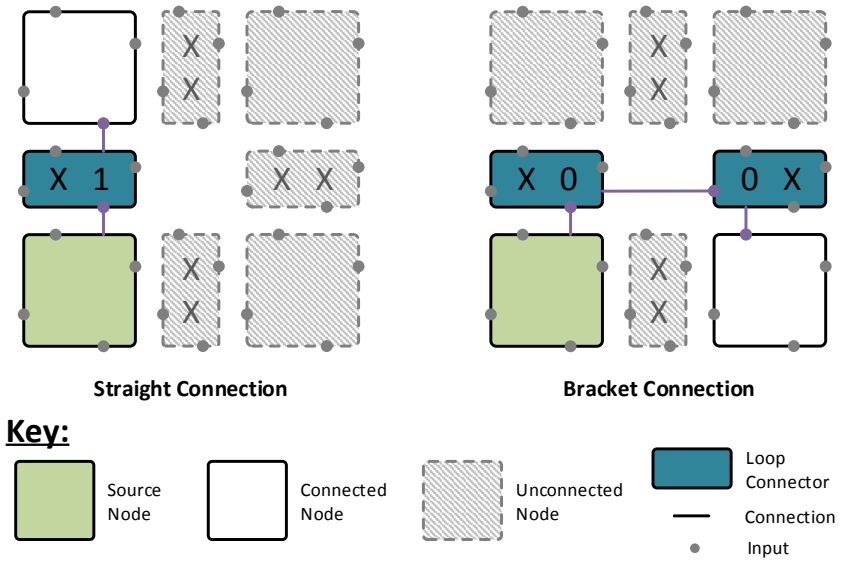


Figure 8.5: Grid architecture connection infrastructure examples. Using pairs of straight connections and two bracket connections, loops are formed within the architecture to connect neurons to one another. Each loop connector block supports three types of connection: **x** - no connection; **1** - straight connection; and **0** - loop end connection.

the loop, passing them along to the next node in the chain. For all other addresses the nodes simply pass the data around the loop, operating in a similar manner to that of shift registers. The nodes use internal registers to determine when to read the current loop values, allowing any node to receive input from any other node within a shared loop. Each node has a virtual neighbourhood that defines which other nodes it may connect to through a single loop, as shown in Figure 8.6. In this fashion, no node is more than one intermediate node away from any other node in the system.

Input/Output (IO) blocks are situated on the grid providing the IO interface for the system. These IO blocks replace nodes within the loops meaning that inputs may be driven locally to any desired loop within the system. Like the traditional nodes, the IO blocks are able to drive data into the loop and read data from the loop, however they also have an external communications point that connects to an external pin on the Integrated Circuit (IC) package. These pins may be connected to sensors or actuators, providing IO for the network; or they may be connected to one another, acting as a long range connection that operates outside of the neighbourhood domain. An example of the grid layout with demo loops is shown in Figure 8.7.

The system clock speed is dependent on the network and may be loosely associated to the largest number of connections that the most connected neuron requires to or from itself. In Figure 8.7 loop 'C' is seen to be the largest connected loop, with 8 units contained within its domain. The system clock must therefore run at least 7 times slower than the internal clock, regardless of the number of neurons used overall. The loops are implemented as a ring of shift registers, with a register located at each node

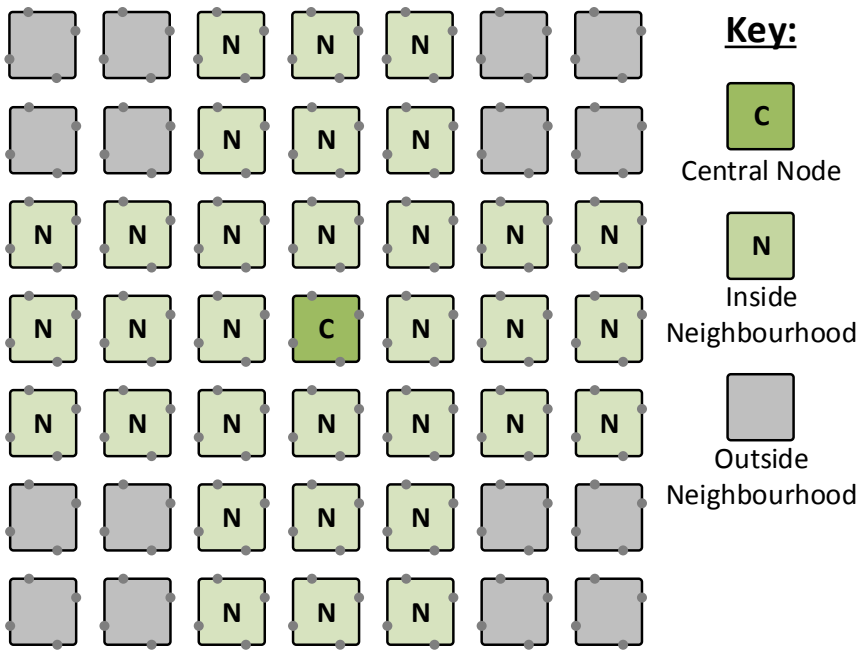


Figure 8.6: Available local neighbourhood of a neuron supported by the grid architecture.

input, meaning that the propagation delay remains fixed regardless of loop size. This architecture can mimic a fixed-width layered network by simply assigning loops down the full length of each column.

Since this system represents a communications solution, the nodes themselves can be defined to use a broad range of different neuron models. In this work, and for simplicity, a configurable Integrate and Fire (IF) model was used. This model contains internal weights that are applied to the incoming stimulus. The weighted input is added together and passed through a threshold block and if the threshold is exceeded, the output is set high ready for the next loop cycle. All loop cycles are kept in synchronisation, meaning that smaller loops shut down once finished while waiting for the larger loops to complete their cycles. This shutdown step for small loops helps reduce the overall power consumption by avoiding unnecessary data transmission.

This grid architecture is configured, as before, using configuration commands. These are sent along a serial control bus that connects to a computer to allow the configuration and monitoring of the system during use. These configuration commands set the connection infrastructure arrangement, the node weights, IO direction and the controllers address space range. In a technique similar to that of FPGA synthesis engines, it is up to the network synthesis tool to arrange the network and fit the neurons into neighbouring node loops, with an aim to minimise the maximum loop size.

Both these architectures require an internal clock and controller that propagates fully through an address space once for each simulated time-step. While the column

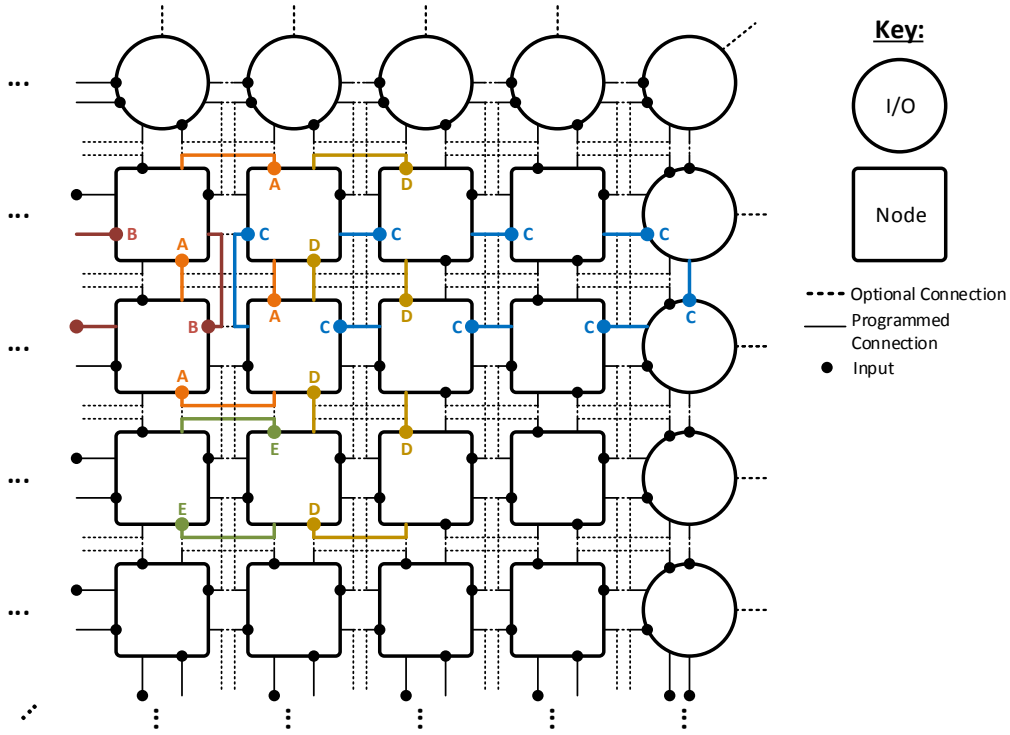


Figure 8.7: Infrastructure of the grid architecture, showing a number of 'loop' configurations labelled A to E. The circular nodes are the IO blocks that surround all edges of the architecture, the square nodes represent the processing, or neuron, blocks. Potential connectivity is indicated by the dashed lines and in the example loops given, actual connectivity is indicated by a solid line.

architecture requires each neuron to have a unique address, the grid architecture requires neurons to only have unique addresses within any given loop. As such, many neurons may drive independent loops at the same time, reducing the total cycle count required per simulated time-step. This architecture supports two-dimensional scaling which is independent of the system clock timings. A pseudo-global connection can be achieved, with any node located a maximum of one node away from any other node, and a single neuron can send and receive in four directions at the same time allowing for rapid transmission through a network. This architecture can support locally connected networks with a small number of global connections.

Intermediate Representation

In order to produce a scalable and practical architecture it is important to consider how a particular network may be synthesised onto the hardware. There are a number of different methods for high-level representation of a network. An intermediate translation must be defined to allow a user to move from such high-level descriptions to a physical mapped solution and this often reflects the physical architecture through its defined structure.

The grid architecture includes a scan chain for loading parameters into the system, thus the intermediate representation will be structured in a serialised form, with the whole network defined as a single entry. To define a network, the connectivity and individual neuron parameters must all be included in the final serialised format. A number of different connection options were shown in Figure 8.5. Each node can support four connections, with one starting on each face of the node. The direction of these connections is determined by the connection infrastructure hardware, forming straight and bracketed connections into larger loops. The selection of these directional connections is performed using single bit parameter values. Alongside the connection information, the serialised data also includes the weights, neuron and IO block settings.

The implications of the interconnect options offered by this architecture must also be considered when mapping a high-level design, and inherent to this is the concept of locality between neurons. Figure 8.6 shows a neuron (the 'central' node) and its local 'neighbourhood' neurons. For two neurons to be connected they must exist within the one-another's neighbourhoods and it is therefore up to the synthesis engine to find an arrangement of nodes which results in all nodes sitting within the neighbourhoods of their connected nodes. This task becomes a fitting problem, similar to that performed by FPGA synthesis engines. Neurons sitting on the same row or column will share in the row or column neighbourhood respectively. This can be used to create layers of neurons, allowing a simple implementation of Feed-Forward Neural Networks (FFNNs).

High Level Network Design Tool

It is necessary to provide some form of high level programming interface to easily map a known neural network onto the grid architecture. For this purpose a software tool has been developed that permits the user to define a network and its corresponding weights. This software tool exports a low level programming file that is used to configure the hardware grid architecture.

The software tool is divided into three core parts. These are the global controls, such as reset and synchronisation; the interconnectivity definitions, used to arrange to loop connections within the architecture; and the neuron configurations, defining the threshold and weights. The global controls are pre-defined by the software tool, ensuring that the reset, write enable and synchronisation signals are all generated correctly.

The interconnectivity definitions are performed using a graphical interface. Each neuron within the network is connected to four interconnect switches in a North, East, South, West fashion. The grid architecture has a rectangular structure and so the configuration process is split into columns, where each column starts and stops with an IO block. Each column is responsible for the interconnect switches on the eastern and northern sides of the neurons with an additional interconnect switch north of the bottommost IO block. This is shown in Figure 8.8. The green cells in interconnection blocks are configured graphically by selecting a routing direction. Only the cells marked in green require definition, with the remaining cells auto-populating to form closed loops.

Each neuron has three standard configurable parameters, the input weights, the input bias and the threshold. The bias and threshold values are singular for each neuron block, whereas the weights are a vector with an entry for each possible connected input. The neuron parameters are setup as shown in Figure 8.9 with 5 neurons in this example numbered N0 to N4 and permissible weights in the range -4 to +3.

This graphical interface makes configuring the grid architecture a straightforward task for the user, greatly reducing the time required to setup and use the proposed neuromorphic solution.

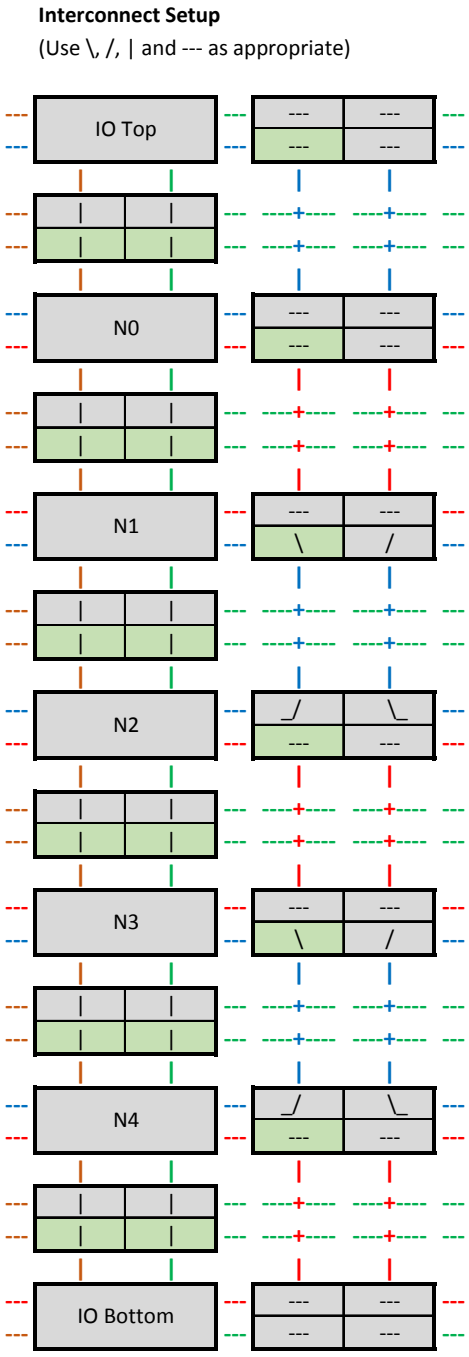


Figure 8.8: Design of the interconnects between neurons within a single column of the grid architecture. In this case there is a top and bottom IO block and 5 neuron blocks N0 - N4, with routing blocks located in-between each neuron and IO block vertically as well as on the right horizontal. The routing blocks can be configured graphically by selecting a routing direction in each of the green cells. The right horizontal routing blocks appear on every grid column with the exception of the far right IO column.

Top Connection Weights

(-4 to 3)		From													
		N + 13	N + 12	N + 11	N + 10	N + 9	N + 8	N + 7	N + 6	N + 5	N + 4	N + 3	N + 2	N + 1	Self
To	N0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N4	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Right Connection Weights

(-4 to 3)		From													
		N + 13	N + 12	N + 11	N + 10	N + 9	N + 8	N + 7	N + 6	N + 5	N + 4	N + 3	N + 2	N + 1	Self
To	N0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N4	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bottom Connection Weights

(-4 to 3)		From													
		N + 13	N + 12	N + 11	N + 10	N + 9	N + 8	N + 7	N + 6	N + 5	N + 4	N + 3	N + 2	N + 1	Self
To	N0	0	0	0	2	0	0	0	2	0	0	0	0	0	0
	N1	0	0	0	0	1	-4	0	0	1	0	0	0	1	1
	N2	0	0	0	0	2	0	0	0	0	0	0	0	0	0
	N3	0	0	-4	0	0	0	0	0	1	0	0	1	1	1
	N4	0	0	0	0	0	0	2	0	0	0	0	0	0	0

Left Connection Weights

(-4 to 3)		From													
		N + 13	N + 12	N + 11	N + 10	N + 9	N + 8	N + 7	N + 6	N + 5	N + 4	N + 3	N + 2	N + 1	Self
To	N0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	N4	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8.9: Grid architecture weight vectors for single column comprising 5 neuron blocks. The weights are in the range -4 to +3 and are configurable for every possible input connection to the neuron.

8.3 Validation Methodology for a Fully Local Architecture

A well constrained and understood biological system is required to assess the applicability of the novel grid architecture design and compare its performance with that of the previously established column architecture. The *C. Elegans* locomotory system, responsible for locomotion generation within the *C. Elegans* nematode, is well documented and therefore this locomotory system was used to demonstrate and validate the representation capabilities of the grid architecture at a network level. The locomotory system of the *C. Elegans* has already seen common usage in both the MBED Cellular Automata Neuron model by Claverol et al [137] and the column architecture by Bailey [15]. These models showed that an ANN was capable of generating the same motor neuron patterns as had been observed in living *C. Elegans*.

This section details the *C. Elegans* event-based locomotive model before discussing the implementation of the model on the novel fully-local grid architecture, allowing direct comparison between the grid and column architectures. The resulting neural network is also used to drive a 2D and 3D model to provide a visual representation of the simulated animal's resulting rectilinear locomotion.

8.3.1 The *C. Elegans* Locomotive Model

First introduced by Claverol [146], the *C. Elegans* event-based locomotive model simulates a small part of this animals connectome, containing only 86 neurons and 180 synapses. This neural circuit is responsible for the forward and backwards locomotion as well as coiling behaviour. The structure of this neural circuit is highly regular with very few variations from animal to animal within the species.

The model may be divided into 10 segments to match the animals nervous system itself. The majority of these segments take the form shown in Figure 8.10, where there is a high level of locality within the model. The muscle cells, denoted DM and VM, drive muscle activation within their respective segments. These are triggered by the motor neurons in adjacent segments, labelled VA, VB, DA and DB. These motor neurons require concurrent activation from both the muscle cells and the synchronization cells, marked AVA and AVB. The two remaining neurons, VD and DD, operate as inhibitory neurons and are responsible for the contralateral inhibition of the muscle on the opposite side of the animal. In this arrangement only a limited number of connections join to adjacent sections and a total of two neurons (AVA and AVB) are connected globally, joining to all segments. The head and tail segments have two additional neurons each (marked NRD, NRV in the head and TSD, TSV in the tail), to act as stimulation points for the rest of the locomotive system. These extra neurons may be seen highlighted in

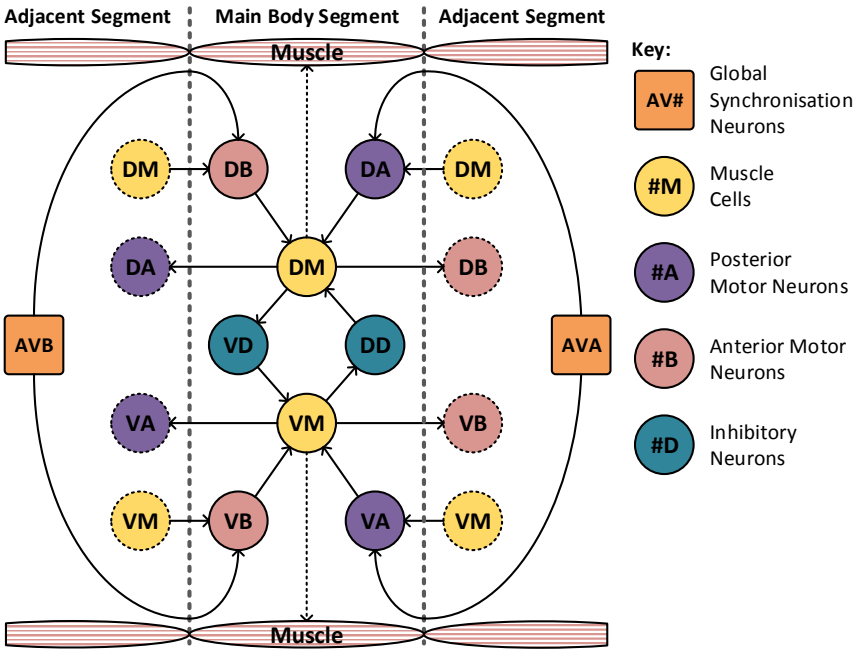


Figure 8.10: Slice of the *C. Elegans* locomotion model originally developed by Claverol [137], showing clear segmentation with local connectivity and sparse global connectivity.

green in Figure 8.11, which shows the construction of the head and tail sections of the locomotive model. Each of these stimulation neurons represent separate neural systems which, combined, guide the animals locomotion. In the event-based model these neural systems are replaced with a set of frequency controlled oscillators that generate the desired stimulus input frequencies. The 8 distinct regions of muscle shown in Figure 8.2 are grouped into two sets: the dorsal set; and the ventral set. These are then treated as a singular muscle with just one dorsal and ventral motor neuron driving each set per segment, as shown in Figure 8.10.

This event-driven model has been utilised in numerous different implementations for neural hardware [146, 147, 148, 149]. These models provide resultant behaviours that new systems may be compared against to ensure functional equivalence and make the locomotive model a practical test for new biologically inspired reconfigurable architectures, providing a simple yet functionally verifiable network inspired by nature.

8.3.2 Representation Validation - The *C. Elegans* Locomotive Model

The locomotive model of the *C. Elegans* may be divided into small, repeated segments (as shown in Figure 8.10). Each segment exhibits a high order of local connectivity, with relatively sparse connections from one segment to another.

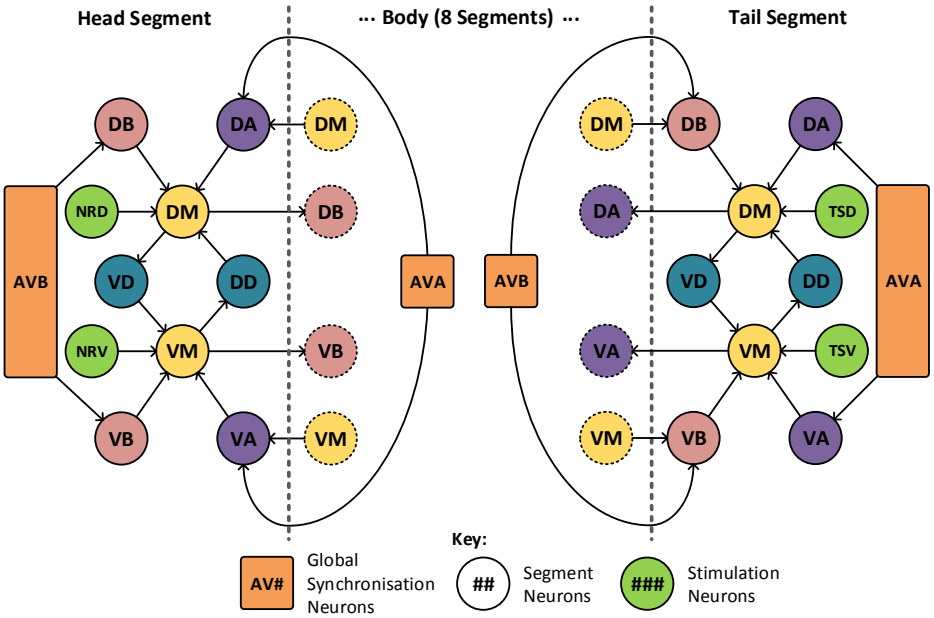


Figure 8.11: The C. Elegans locomotive head and tail segments - showing high locality, with limited connections to adjacent segments. In total, there are only two global neurons that connect to all segments. These global neurons (AVA and AVB) are shown duplicated here for readability.

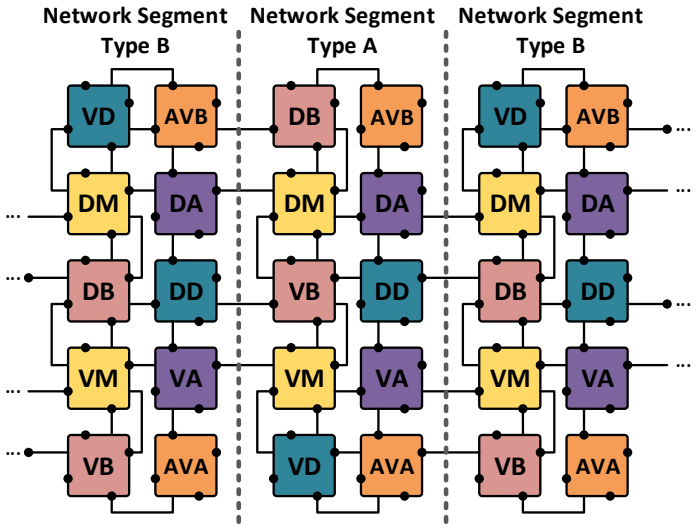


Figure 8.12: The segment from Figure 8.10 implemented on the grid architecture with full scalability, extending the model to an arbitrary number of segments is a simple case of inserting more columns, no changes to the fundamental architecture are required.

By utilising the local connectivity seen within this model, it was implemented using the grid architecture system described in Section 8.2.2 (an example mapping is shown in Figure 8.12). The largest loop within this implementation, which lies along the width of a single segment, contains 10 neurons. This system therefore requires 9 clock cycles per simulated system time-step regardless of the number of segments implemented. This represents a 9× speed increase in the 10 segment system when compared to the direct implementation of Bailey [15].

The sensory AVA and AVB neurons are actually two single neurons that connect along the entire length of the *C. Elegans*. These neurons have no incoming connections from within the network and they may therefore be divided into a number of identical neurons that are all driven by the same external stimulus, as shown in Figure 8.12.

The *C. Elegans* locomotive model was implemented on the grid architecture using a CycloneV FPGA. With the reconfigurable grid architecture built onto the FPGA, configuration files were generated to describe the locomotive models network. While such configuration files would nominally be generated by a synthesis engine, the ones used in this test were defined by hand. This was done to ensure that any comparisons drawn between the grid and column architectures themselves were tested fairly without external influence from synthesis algorithms or hidden layout decision processes. Synthesis and system fitting are two concepts well established within FPGA and Custom IC development and therefore are of limited interest when verifying the architectures fundamental ability to represent a problem or task.

HDL test bench files were developed and used in Modelsim, a multi-language HDL simulation environment by Mentor Graphics, to program the grid architecture with the generated configuration files. This test bench also simulates the stimuli for the network, recording the response at each time step. It was designed to wrap around the architecture-under-test, connecting to the IO blocks and the control/programming interfaces. This allowed the architecture to be reconfigured on command, providing monitoring for the output signals and direct control over the input stimuli. The stimuli for generating forwards, backwards and coiling behaviour within the locomotive model are shown in Figure 8.13. Each locomotive behaviour is selected by generating the associated oscillation signals at the AVA, AVB, DRN, DTN, VRN and VTN neurons as shown. The test bench was configured with custom control signals to easily select and generate the desired stimuli on command. These input stimuli each yield a predictable motor response, and therefore the test bench was setup to record all motor neuron outputs for verification. These recorded motor neuron signals were then written to file to support further investigation and validation.

The test bench was configured to test for forwards, backwards and coiling behaviour as seen in a healthy animal. Since the system is easily configurable, gene knockout tests may also be performed. In gene knockout tests, specific neurons or synapses are disabled. The resulting network operation (in this case motion generation) is then re-tested and compared against a healthy sample. These tests are often used to help

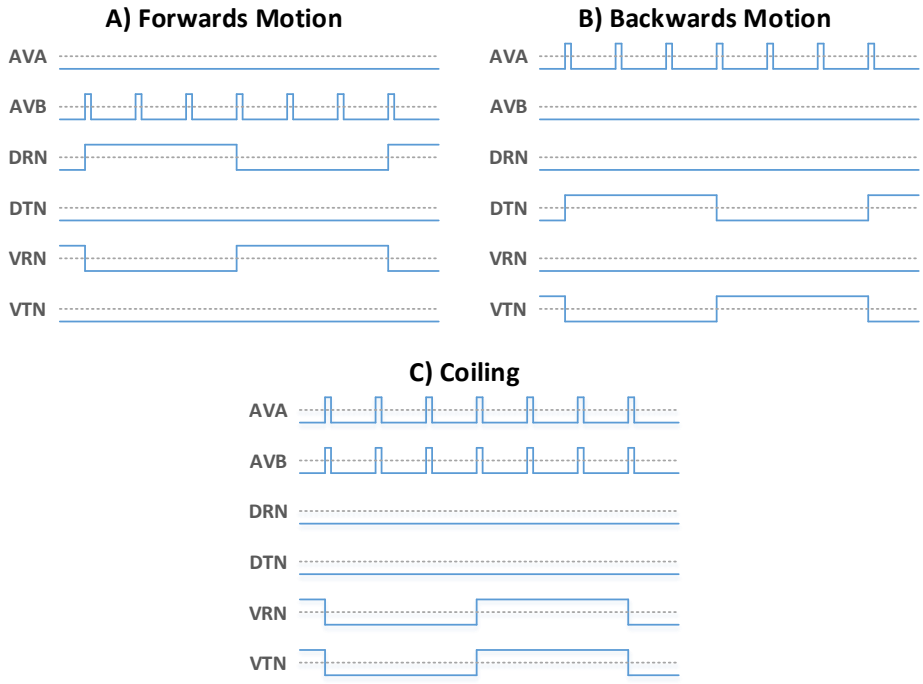


Figure 8.13: The stimulus signals passed to the head and tail control neurons and the two global AVx neurons. Signals of the form A) produce forwards motion, signals of the form B) yield backwards motion, while signals of the form C) yield coiling behaviour.

inform research into the specific role and operation of a given neuron within the nervous system. These tests provide an interesting comparative metric when assessing new artificial implementations, as they are usually performed on live biological specimens and therefore yield biological data against which the artificial system may be compared. For example, the UNC25 gene knockout mutation causes the Dorsal and Ventral motor neurons to latch into a constant firing mode when activated. This causes the animal to seize up, resulting in a shrinking behaviour.

Through the implementation of a healthy *C. Elegans* model, alongside the UNC25 gene knockout model, the grid architecture may be validated against both the hybrid-local column architecture and biological results.

8.3.3 2D Mechanical Model

In order to further validate the architectures output, a new 2D mechanical model of the *C. Elegans* was developed. This model reads the output muscle signals generated by the implemented model, driving the muscles on the simulated animal to generate motion. This motion may then be compared against recordings of the animal itself providing insight into the resulting locomotion caused by a given signal train.

Since the *C. Elegans* is a soft-bodied creature, this mechanical model must act to

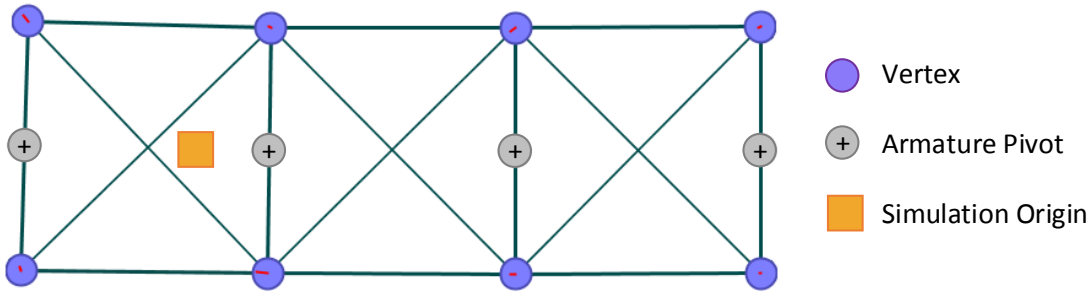


Figure 8.14: Screen capture of the mechanical model, showing the vertices as purple circles; the armature joints as grey circles; the muscles and structural supports as lines and the simulation origin as a yellow square.

maintain the animals structure while also applying deformation in event of muscle activation. This model was built using principles from force-directed graphs. As shown in Figure 8.14, vertices were placed at equal distances along the length of the *C. Elegans*, conceptually representing the ends of the muscles. Edges were drawn between neighbouring vertices along the length of the animal, representing the muscles responsible for locomotion. Finally cross-bars and vertical struts were added to ensure that the simulated volume maintained its structure when force was applied by the muscles.

Edge Forces (Muscles and Structure)

Each of the edges are treated as linear springs with non-zero resting length, l_0. Connected vertices are therefore attracted to one another using Hooke's law as shown in Equation 8.4, where F_s is the force applied to a vertex that is connected to another vertex by an edge of length l with a spring constant k_s. The unit vector $\hat{u}$ points from the vertex acted upon towards the vertex connected by the edge under inspection.

$$\vec{F_s} = k_s \cdot (l - l_0) \cdot \hat{u} \tag{8.4}$$

While all edges operate as spring attractors, they may be conceptually separated into two categories. The first class are those that operate as muscles and sit along the two sides of the animals; the second are those that act as structural supports for the model. These structural edges maintain a fixed resting length throughout the entire simulation, acting to support and shape the body of the simulated *C. Elegans*. The edges marked as muscle, however, are modified by muscle contraction, with activation causing the resting length, l_0, of the edge in question to shorten. Muscle relaxation causes the length to restore to its default value.

The muscle length modification is not instantaneous, using a simple feedback loop to control the length at any simulation time step. This helps ensure the motion is fluid and is a closer representation to the way muscles contract in biological systems. This feedback loop is shown in Equation 8.5, where $\bar{A}$ represents the muscles activation signal (0 when activated and 1 when relaxed), l_{max} is the relaxed resting length of the muscle, l_n is the current length of the muscle, l_{n+1} is the next muscle length, and k_m is the muscle constant used to scale the muscles response to stimulus.

$$l_{n+1} = (\bar{A} \cdot l_{max} - l_n) \cdot K_m + l_n \tag{8.5}$$

Vertex Forces (Volume)

The vertices of the model are repelled from one another to ensure that the body doesn't collapse during muscle contraction. This force is calculated using an inverse square law to ensure that its effect is minimal when vertices are adequately spaced. In this simulation a form of Coulomb's law was used to provide this vertex repulsion, as shown below in Equation 8.6:

$$\vec{F_{vi}} = -K_e \cdot \frac{q_1 q_2}{x^2} \cdot \hat{v} \tag{8.6}$$

where k_e is a constant, q_1 and q_2 are virtual charges to control a vertex's influence on the model, d is the distance between the two vertices in question and $\hat{v}$ is a vector which points towards the other vertex in the vertex pair under consideration. This inverse square law ensures that the repulsive forces become significantly larger than any other force within the system as the vertices near one another.

Combining Forces

The law of conservation of momentum may then be applied to calculate the acceleration caused by the superposition of these forces as shown in Equations 8.7 and 8.8. In these equations, the vertices are given a mass, m; the total spring force, F_s and the total vertex repulsion force, F_v are calculated; and the velocity v is used with a drag constant k_d.

$$0 = m\vec{a} + k_d\vec{v} + \vec{F_s} + \vec{F_v} \tag{8.7}$$

$$\vec{a} = -\frac{1}{m} \cdot (k_d\vec{v} + \vec{F_s} + \vec{F_v}) \tag{8.8}$$

This acceleration value may then be used to calculate the new velocity of the vertex at each time step, ensuring that the system finds equilibrium at rest.

This mechanical model was designed to read the raw ventral and dorsal muscle activation signals produced by the grid hardware architecture on the Modelsim test benches. These activation signals were stored by the test bench as time-synchronised boolean values representing motor neuron activations. These motor neuron signals were linked to the associated muscle length using Equation 8.5.

Once the simulated motion was generated, the angles made between the *C. Elegans* segments is calculated and saved. This data makes it possible to support 3D model actuation as outlined in Section 8.3.4.

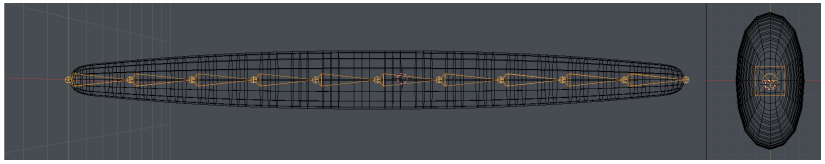
8.3.4 3D Model

A 3D model was also built to allow direct visualisation and comparison of the animals movement. This simulation makes validation of the neural model easier as it provides a visual representation of the results that may be compared against the biological counterpart.

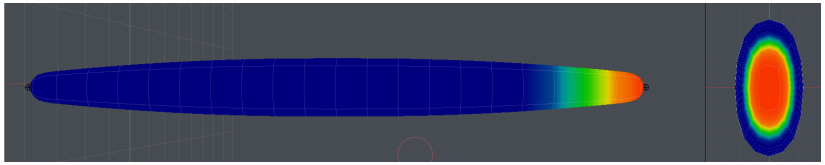
This model was built using a 3D modelling and animation package [1] and rigged with a simple linear armature, shown in Figure 8.15a. Each bone in the armature represents a segment along the animals length. Vertex weights were used to control each bones influence along the simulated animals length, shown for the head bone in Figure 8.15b. The segment angles generated using the 2D physics model were imported and applied to the armature using a simple python script. Finally materials, backdrops and lighting were added to the simulation, allowing the resultant motion to be simulated and rendered in 3D.

With the pipeline from hardware implementation to 3D model designed, it is trivial to trial new network modifications, such as gene knockout mutations - resulting in a visual 3D representation of the animals motion that may be quickly compared against biological findings.

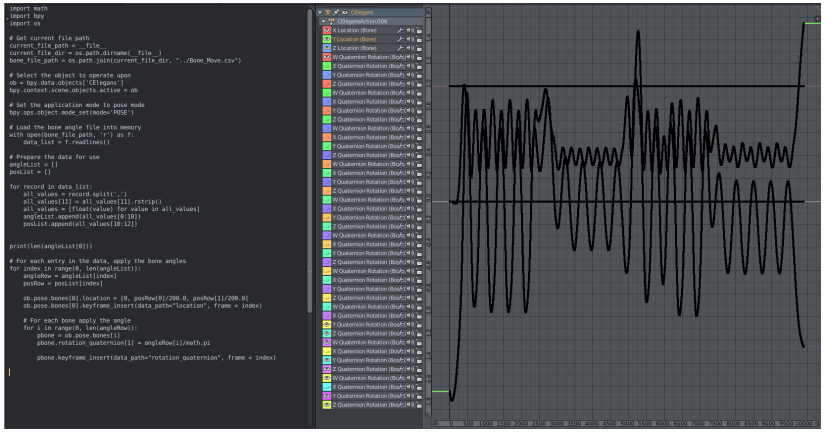
[1] Blender was used for this work, available: `http://www.blender.org`



(a) Armature and body of the simulated *C. Elegans* model, with the armature highlighted in orange.

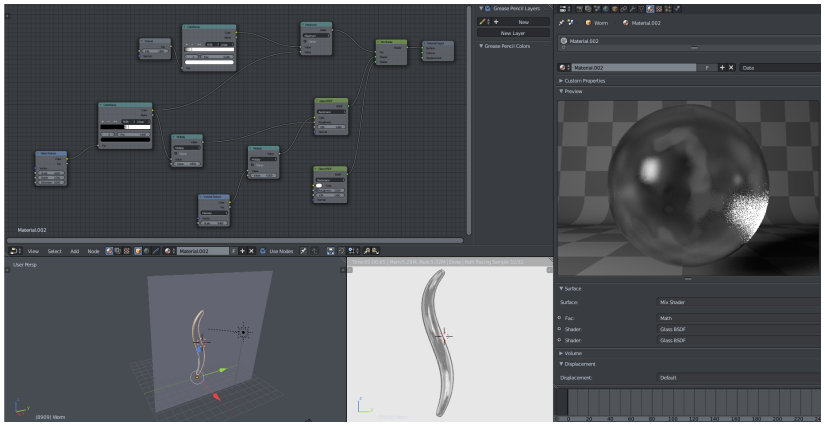


(b) Weight painting example to control the influence of each armature bone.



```
import math
import bpy
import os

# Get current file path
current_file_path = __file__
current_file_dir = os.path.dirname(__file__)
bone_file_path = os.path.join(current_file_dir, "../Bone_Move.csv")

# Select the object to operate upon
ob = bpy.data.objects['Elegans']
bpy.context.scene.objects.active = ob

# Set the application mode to pose mode
bpy.ops.object.mode_set(mode='POSE')

# Load the bone angle file into memory
with open(bone_file_path, 'r') as f:
    data_list = f.readlines()

# Prepare the data for use
angleList = []
posList = []

for record in data_list:
    all_values = record.split(',')
    all_values[11] = all_values[11].rstrip()
    all_values = [float(value) for value in all_values]
    angleList.append(all_values[0:10])
    posList.append(all_values[10:12])


print(len(angleList[0]))

# For each entry in the data, apply the bone angles
for index in range(0, len(angleList)):
    angleRow = angleList[index]
    posRow = posList[index]

    ob.pose.bones[0].location = [0, posRow[0]/200.0, posRow[1]/200.0]
    ob.pose.bones[0].keyframe_insert(data_path="location", frame = index)

    # For each bone apply the angle
    for i in range(0, len(angleRow)):
        pbone = ob.pose.bones[i]
        pbone.rotation_quaternion[1] = angleRow[i]/math.pi

        pbone.keyframe_insert(data_path="rotation_quaternion", frame = index)
```

(c) A Python script was used to control the armatures bone positions and rotations, reading the data from the exported 2D model data.



(d) Material, lighting and camera setup used to generate the final motion animations. Selected to match real biological recordings.

Figure 8.15: The 3D model was rigged using a standard armature, before the 2D model data was used to simulate the *C. Elegans* deformation and motion. Materials and and a simple white background were used to match the footage from biological examples.

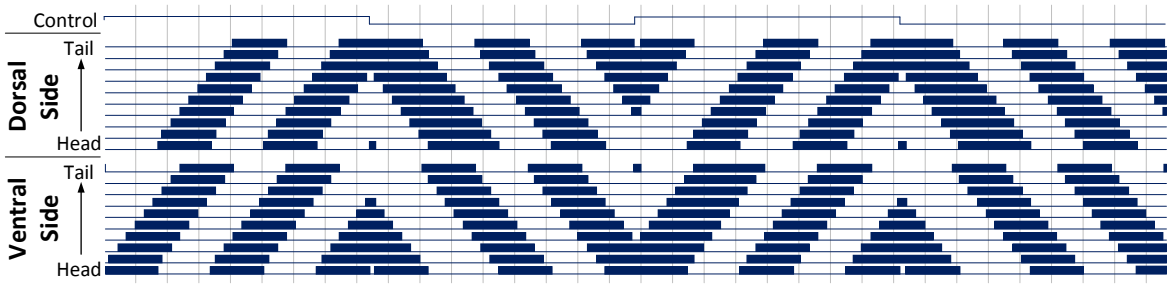


Figure 8.16: Forward and backward locomotion behaviour of a 10 segment *C. Elegans* model, with a bi-phasic control signal to generate stimulus applied to the head and tail of the model. This results in a propagating wave of signals leading to sinusoidal motion in the *C. Elegans*. Muscle activation is shown as a solid block of colour due to the high frequency oscillations of the muscle cells.

8.4 Validation Results

As described in Section 8.3 the reconfigurable grid architecture was verified using the *C. Elegans* locomotive model. First a stimulus was applied that, in nature, is observed to cause forwards and backwards motion. Then the coiling and UNC25 knockout behaviours were also tested. The implemented networks response to this stimulus was recorded and used in a 2D and 3D model to produce a simulation of the locomotive behaviour. This section presents the results from these test, comparing them with previous studies and the biological animal where relevant.

Forwards/Backwards Motion

The grid architecture was configured with a simulated *C. Elegans* constructed from 10 segments. Forwards and backwards locomotion was then observed using a bi-phasic control signal to produce the appropriate head and tail stimulation. The resulting motor neuron signals generated by the grid architecture are shown in Figure 8.16. The top-most trace in this figure is the control signal used to drive the forwards and backwards stimulus. This control signal is setup to oscillate at an arbitrary low frequency, resulting in an alternating forwards and backwards locomotion.

The top bundle of signals in Figure 8.16 form the dorsal muscle cells, while the bottom bundle represent the ventral muscle cells. Activation of these neurons is shown as blocks of colour due to the high frequency oscillations produced by the muscle cells. These neurons are numbered from head to tail starting at 0, such that movement propagating head to tail will move from the bottom to the top of each bundle. Vertical lines are drawn every 500 milliseconds on the x-axis.

Activity begins on the Ventral side of the head and propagates down to the Dorsal side of the tail, taking approximately 2900 milliseconds to complete one full cycle. These signals

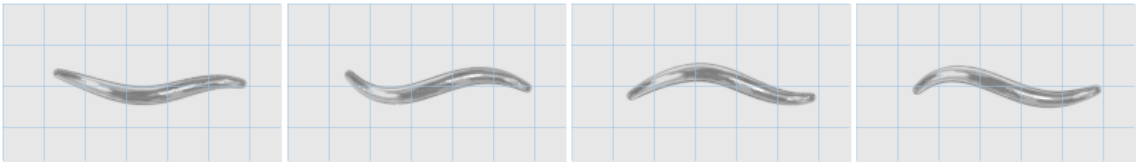


Figure 8.17: Frames taken from the 3D animation[2], showing the sinusoidal motion indicative of forward movement in the simulated 10-segment *C. Elegans*. In this animation the animal is moving left-to-right, as shown by the propagating waves moving right-to-left down the length of the animal. The muscle contraction data for this animation was generated using the grid architecture.

produce a sinusoidal motion in the animal, yielding a forwards locomotion. Figure 8.17 shows frames from the animated 3D model, where this propagating sinusoidal motion is shown with the muscle activation propagating right-to-left down the length of the *C. Elegans*.

These results were consistent with Bailey's column architecture results and yield the same propagating waves as seen in previous work and observed in nature [15]. The muscles were observed to oscillate at a frequency of $0.57Hz$ which matches the findings of Karbowski *et. al.* [150]. The propagating muscle signals were found to match those of Claverol *et. al.* [146] in both shape and structure. The exact timing in the signals varies from model to model, largely due to differences in the timing of stimulation pulses. The works of Mailler *et. al.*, Niebur *et. al.* and Bailey *et. al.* [151, 152, 147, 148] are all agreed on the underlying nature of propagating muscle signals and the resulting serpentine motion. These properties are observed in the muscle activation signals moving down the length of this simulated animal and in the 3D model itself. Boyle *et. al.* additionally provide stills from simulations of their model [153]. These stills were compared against the 3D animation and found to have the same core locomotive properties.

The locomotive model was mapped onto the grid architecture with a maximum loop size of 10. This means that only 9 sub-steps were required to pass all neurons outputs to their neighbours within the system and preform a single simulation step. In contrast the hybrid-local column architecture required all 86 neurons to drive the buses once each for a simulation step meaning that 86 sub-steps were required. This reduction in sub-steps enabled the grid architecture to outperform both the column architecture and previous implementations by a simulation speed increase of 9× and allows new segments to be added to the locomotive model without any reduction in performance.

[2]Full video available at: `https://youtu.be/PKIIKkSBtjE`

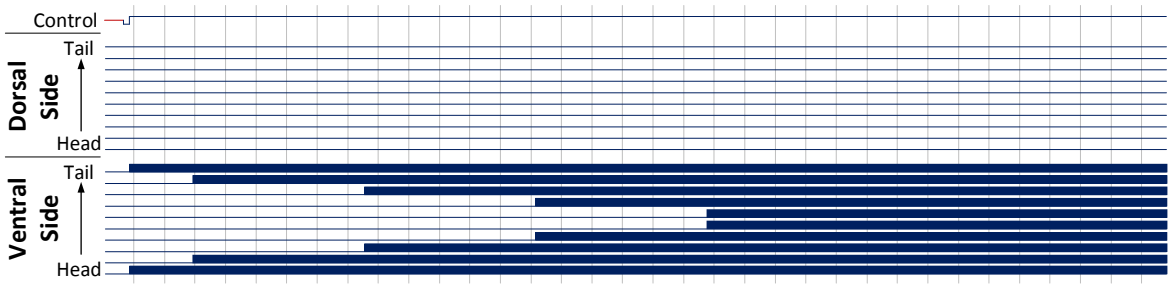


Figure 8.18: Coiling behaviour of a 10 segment *C. Elegans* model, with stimulus applied to the Ventral side of both the head and tail. This results in motor neuron activation along the ventral side only, producing a coiling action towards the centre. Muscle activation is shown as a solid block of colour due to the high frequency oscillations of the muscle cells.

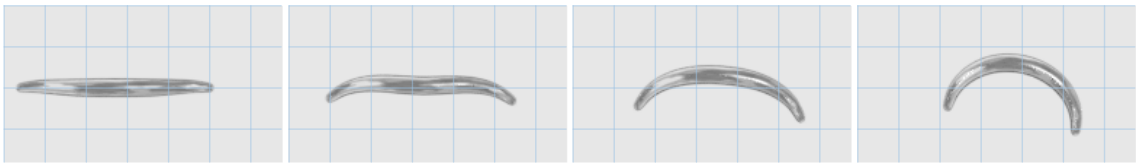


Figure 8.19: Frames taken from the 3D animation, showing the coiling behaviour in the simulated 10-segment *C. Elegans* on the grid architecture, with time progressing from left-to-right. The muscle contraction data for this animation was generated using the grid architecture.

Coiling Behaviour

Coiling behaviour has also been demonstrated in previous implementations of the *C. Elegans* locomotive model. Figure 8.18 shows the Dorsal and Ventral muscle cells response to a stimulus applied to both the head and tail simultaneously. In this situation, activation begins at both the head and tail of the animal and then propagates to the centre. This muscle activation leads the animal to coil towards the stimulus, which in this case was applied to the Ventral side of the animal. The results for this test are consistent between the grid and column architectures. The 3D model was again generated, showing that the signals do indeed lead to coiling behaviour towards the stimulus, as observed in nature. Frames from this simulation may be seen in Figure 8.19.

UNC25 Knockout

The UNC25 gene knockout has a well defined impact on the operation of the *C. Elegans* locomotive system. This mutation disrupts the pathway for synthesis of the inhibitory neurotransmitter GABA [154]. This disruption is reflected in the class D neurons (DD and VD) which become unable to release inhibitory neurotransmitter onto the muscles

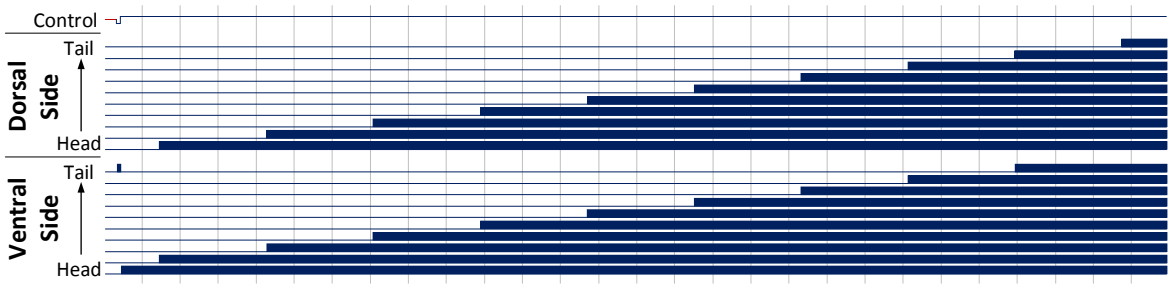


Figure 8.20: UNC25 knockout behaviour of a 10 segment *C. Elegans* model, with stimulus applied to the head and tail of the model to generate forwards motion. This results in a seizure in the animal, with both dorsal and ventral muscles firing as the activation propagates down its length. Muscle activation is shown as a solid block of colour due to the high frequency oscillations of the muscle cells.

to stop them firing. In this mode, the muscle cells enter a state of constant firing, resulting in a full muscular seizure along the animals length [149].

Within the simulated locomotive model this mutation may be implemented by significantly increasing the firing threshold of the DD and VD neurons. Applying the standard forward motion stimulus to the head of a *C. Elegans* with this gene mutation leads to a propagating seizure starting at the head. Figure 8.20 shows the motor neuron response to such stimulus. As expected the mutation results in seizures in the model on both architectures, propagating from head to tail as the locomotive behaviour communicates down the animals length.

Summary

The grid architecture has been shown to fully emulate the *C. Elegans* locomotive model with no failure or shortcoming in its network representation. The timing and structure of muscle activation waveforms were checked against previous works and found to be in accordance with the new results. The 3D model behaviours were also reviewed visually and found to be consistent with observed behaviours seen in nature. Forwards, backwards and coiling behaviour were all demonstrated and the system was shown to be suitable for testing gene knockout behaviour with accurate and biologically correct results.

8.5 Results Analysis

The *C.Elegans* model has been successfully demonstrated on the new fully-local grid architecture. An important aspect to consider is the scalability (and its effect on speed) that is possible with this architecture when compared against existing systems.

Table 8.5: Worst-case maximum frequencies for the key column and proposed grid architecture subsystems.

Sub-system	Column Architecture	Proposed Architecture
Neuron Clock	9.84 MHz	82.13 MHz
Communications Clock	187.34 MHz	82.13 MHz

While the column architecture benefits from supporting a hybrid communications infrastructure, its clock speed is positively correlated to the neuron count. The grid architecture, however, is constrained by the maximum loop size required when fitting the network to the underlying system. Since the 10 segment C.Elegans requires 86 neurons and the largest grid loop was 10 neurons, the grid architecture should be 8.6× faster than the column architecture when implementing the same locomotive model.

This performance gain assumes that both systems run with the same internal clock speed. In reality this is not possible, as different architecture modules will have different clock constraints. The Altera tool-suite provides a timing analysis tool that may be used to calculate the clock constraints for each module. The key clock constraints for both the column and grid architecture are shown in Table 8.5. These maximum clock frequencies were generated using the synthesis timing analysis tools. The grid architecture neurons latch the communications loops, meaning that the communications clock is limited by the neuron clock rate. The column architectures neuron clock is also notably slower than the other modules in this table. This difference is due to the way synapses may be chained together within the architecture and represents the absolute worst-case speed, where all synapses are chained together resulting in a large propagation delay. Taking these delays into consideration the grid architecture is 3.5× faster than the column architecture when implementing the 10 segment locomotive model.

While the column architecture supports a faster communications clock, the grid architecture benefits greatly from the reduced number of neurons on each communications bus as a direct result of its locally connected nature. This means that large networks require fewer communications clock cycles to simulate a single system time step. With the grid architecture, each communications loop that is isolated from the remainder of the communications network increases the overall simulation step frequency, improving the speed at which simulation results may be produced. At this stage, the neuron clock speeds become the dominant limiting factor. The column architecture's neuron clock speed closely correlated to the number of interconnected synapses implemented within the simulation. The grid architecture, however, supports an inherent internal latching of the data on each loop resulting in a fixed neuron clock speed. This fixed frequency means that the grid architecture's neurons achieve a speed improvement of around 8× that of the column architecture's worst-case neuron speeds.

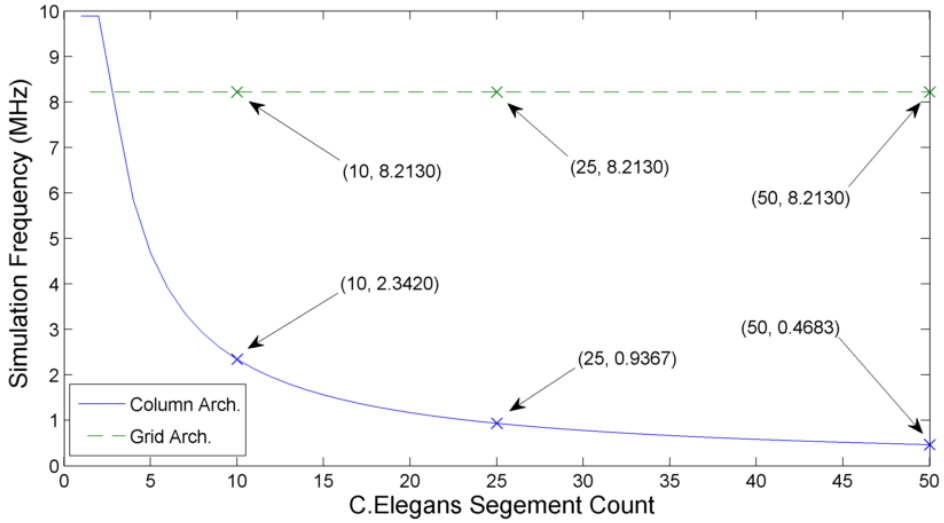


Figure 8.21: Comparison of the column architecture and grid architecture simulation time-step frequencies with increasing network size. Each additional segment results in a decrease of the hybrid-local column architecture, while the fully-local grid architecture maintains it operating simulation step frequency of 8.213 MHz.

The impact of system size on system speed may be calculated by increasing the number of segments used in the *C.Elegans* model. As shown in Figure 8.21, the column architecture forms a 'roof-line' model, where suitably small networks (such as a 1 segment *C.Elegans*) are limited by the neuron clock frequency. With the addition of each segment, the total number of neurons simulated is increased by 8. This increase in neuron count causes the simulation frequency to fall away asymptotically towards zero, with the communications frequency now limiting the overall speed.

The grid architecture has a constant simulation frequency as the largest loop size does not increase with the addition of more segments. In Figure 8.21, the 3.5× gain may be seen in the 10 segment simulation while the estimated 8.6× speed improvement occurs when simulating a 25 segment model. Since the hybrid-local column architecture continues asymptomatically, the simulation frequency gain of the grid architecture continues to increase as new segments are added.

This demonstrates how locally constrained architectures can offer significant benefits when simulating systems which also represent a certain degree of locality within their structure. Indeed a 50 segment *C.Elegans* simulation realises a simulation speed improvement of as much as 17.5× when implemented on a fully-local architecture.

8.5.1 Suitability of Fully Local Systems

Having shown that the fully local grid architecture is capable of representing a biologically inspired network, it is important to now consider whether such systems are

practical for such tasks. In many cases this comes down to a subjective choice based on software preferences, hardware availability and ease of application. It is possible, however, to identify four fundamental criteria that may be used when selecting a suitable hardware system for biological emulation. These criteria are biological accuracy, scalability, performance (which could be in terms of speed, area or both) and flexibility. For each of these criteria, the proposed grid architecture will be compared with the previous hardware implementations of Claverol [137] and Bailey [15].

Biological Accuracy

As has been demonstrated in section 8.4, the basic locomotory behaviour, coiling response and the effect of UNC-25 knockout has been validated in the model implemented using the grid architecture. These results are consistent not only with the observed biological behaviour, but also with the hardware models developed by Claverol [137] and Bailey [15]. There are more mutation conditions that could be evaluated however these three fundamental tests are validation of the efficacy of the model as they show the independent neuron behaviour under locomotion, the asymmetric behaviour of coiling and the shrinking behaviour of the UNC-25 knockout. This clearly demonstrates the fundamental behaviour of the implemented model is in full agreement with the underlying event-based locomotive model, and so the grid architecture could therefore used as a research tool for biological neural network emulation and real-time experiments.

Scalability

One key advantage of the grid architecture is found in its 2-dimensional nature, resulting in the ability to scale the system in both axes arbitrarily. Using the software programming interface, the design is abstracted to a network level making it very easy to generate and program a network accordingly. The connectivity is assumed to be highly local, and while this will not provide the full connectivity for completely arbitrary networks, long-range connections can be defined using the IO blocks to jump between loops. This is analogous to the approach used in the well known Globally Asynchronous, Locally Synchronous (GALS) systems. Importantly, this feature permits the rapid scaling of models such as *C. Elegans* to an arbitrary number of segments without any change in underlying network architecture.

Performance (speed/resources)

From a resource perspective the grid architecture is an efficient system. Using a level of granularity that maps from the neuron network level directly to hardware, this system offers sufficient local connectivity to ensure that the model utilizes the communications

infrastructure effectively. It has been shown that the grid architecture achieves a significant speed improvement over the column architecture. For locally connected networks, with relatively low numbers of connections, this performance improvement can be dramatic - as in the case for the *C. Elegans*, where the largest loop within the model, which lies along the width of a single segment, contains only 10 neurons.

Flexibility

The abstraction of the model to a hardware architecture makes the whole approach extremely flexible from a network definition and programming perspective. Unlike the fully coded approaches of Claverol and Bailey. The HDL library approach of Bailey meant that a standard synthesis software tool would be required to configure the network, with the resulting design not necessarily representative of the locally connected nature of the nervous system being analysed.

Using the software interface to define the connectivity and the weights makes the process straightforward, and most importantly the definition of an underlying hardware fabric gives a known structure and fundamental behaviour that is consistent and simple to extend or modify. This solution is more akin to the operation of FPGAs, where a well defined hardware system supports abstracted circuit definitions to allow rapid experimentation and implementation.

8.6 Limitations of a Locally Connected Architecture

The results in Section 8.4 show that a predominantly local architecture, such as the grid architecture, is capable of simulating a naturally occurring network, with a biologically feasible response to gene knockout mutations. Since there are resource benefits to utilising locally connected architectures, it is also of interest to consider how artificial networks map onto such systems.

One of the most popular neural network topologies at present is the CNN; typically CNNs are used for pattern recognition problems [5]. These systems are structured as deep feed-forward networks, with layers of shared weights to convolve filters across the whole input space. The layers in a CNN are typically well regimented, with connections only present between conceptually neighbouring layers. These layers fall into three key categories: the convolutional layers, which perform a convolutional operation on the input data; the pooling layers, which combine outputs from a layer, compressing the data into a smaller dataset; and the Multi-Layer Perceptron (MLP) layers which perform the final classification task. These MLP layers are typically located at the output side of the system, operating on the filtered data. CNNs use the principle

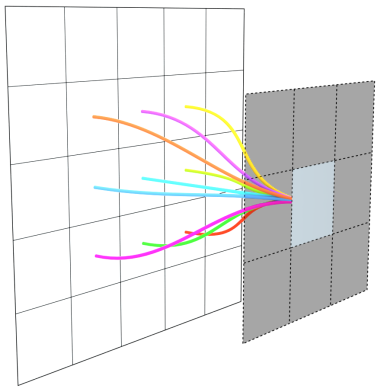


Figure 8.22: Example receptive field of a single pixel in a convolutional layer. Each pixel takes input from a local set of points on the previous layer.

of shared weights within their convolutional layers which dramatically reduces the training times and problem space exploration requirements. Each neuron in these layers receives input from a sub-sample of spatially local data points known as the receptive field (shown in Figure 8.22). The neurons within the layer therefore output a filtered result that is equivalent to convolving a single neurons receptive field across the entire dataset.

With both the clear layered structure and neuron receptive fields, it is apparent that there is a significant measure of locality within CNNs. However, when attempting to map such networks to the grid architecture it was found that CNNs scaled poorly. With 2D image recognition CNNs, the output of each convolutional layer produces a filtered 2D image. These images are conceptually stacked within the same space, with one image layer producing multiple filtered feature layers/images. This stacking must align along a new dimension to keep the convolutional neurons spatially near to their receptive field, meaning that a 3-dimensional structure is required to represent a CNN for 2D image recognition.

The extra dimension required to represent a CNN creates a serious fitting problem when attempting to fit a 2D image recognition CNN onto a locally connected hardware architecture. The architectures are constructed using standard IC fabrication technologies that are not continuously scalable in the third dimension, supporting only a small number of discrete layers. As such, the local communication infrastructure in such systems quickly reaches capacity when unpacking a locally yet densely connected three-dimensional CNN to fit onto a fundamentally two-dimensional architecture.

The scale of modern CNN implementations also causes issues when attempting to produce a hardware accelerated solution. Previous implementations have resorted to splitting the convolutional layers apart, calculating each one sequentially and then storing the working results in off-chip DRAM for further calculations [155, 156, 157, 158].

Even these measures result in layers that cannot be fitted to a modern FPGA system and in each system a further tiling operation must be performed to break the layers into smaller manageable parts. Once the CNN has been successfully defined in a recursive manner further acceleration may be achieved by systematically exploring the impacts of loop unrolling, loop pipe-lining and tile sizing. These methods are described in detail by Zhang *et. al.* and shown to yield a $\sim 2\times$ improvement over previous recursive implementations [155]. Such systems again differ massively from the CNN structures seen in nature, with the recursive approach requiring significant communications and data manipulation. Data reuse becomes a key aspect in such accelerators, attempting to reduce this communications dependency by reducing the memory access required by the system [157]. The proposed grid architecture supports reconfiguration and could therefore be utilised in a recursive manner, however the dimensionality issues previously mentioned would cause a significant performance decrease. Many CNN accelerators are designed specifically for CNN implementations and therefore utilize architectures and innovations that yield significant power and speed advantages over implementations of CNNs on generic neuromorphic systems.

It is worth noting that this dimensional issue was not met with the *C. Elegans* locomotive model. However this is largely because the model was already simplified to constrain the simulated motion onto a plane. The animal itself has a total of eight rows of muscles along its length supporting free motion in 3 dimensions. The locomotive model simulates only two of these muscle rows, limiting the motion to only dorsal and ventral contraction on a surface.

Neuromorphic systems were originally conceived in effort to close the gap between natural biological processing and standard digital techniques [6], however with current fabrication techniques the design of such systems are constrained to a limited number of 2D layers. Biological systems, on the other-hand, operate in dense 3D structures. This loss of dimensionality leads to new challenges, causing artificial systems to require significant communications resource. Recursion and time-division multiplexed hardware resources have found considerable use in neuromorphic systems. These methods provide some mitigation for the dimensionality issue by effectively transferring the lost dimensions into the time domain. Future technologies such as 3D-fabrication techniques may offer new solutions for this shortcoming in artificial systems, however such systems are not readily available at this current time.

8.7 Conclusions

With Moore's Law reaching its inevitable end, a new processing paradigm must be developed. It can be observed that most hardware implementations are large systems with support for full global connectivity. However, many naturally occurring neural systems are locally-connected specialised systems. In this chapter a new fully-local reconfigurable architecture has been described and shown to be capable of simulating real-time biologically inspired neural networks.

The simulation results matched previous work and that of biological observation, proving that such systems are a good option for testing neuron ablation. The resulting $0.57Hz$ muscle activation oscillation during forwards locomotion was shown to be consistent with the work of Karbowski *et. al.*[150]. The reconfigurability offered by the system makes it easy to adjust a network and rapidly iterate on an experiment making it a practical system to complement wet-work experimentation. The locality of the grid architecture was found to provide a maximum clock-speed improvement of between 3.5× and 17.5× that of the hybrid column architecture when simulating a multi-segment C.Elegans locomotive model. This shows the advantages of implementing networks on architectures that reflect the inherent structure of the network within the architectures underlying arrangement.

In addition to the local model, an animated model of a *C. Elegans* was constructed and connected to the neural network outputs, allowing an artificial locomotive model to drive the motion of the simulated animal. This provided a clear way to compare the resulting locomotion with that of biological recordings and other simulations. The results from this model were checked against the stills provided by Boyle *et. al.* and found to be in accordance with their results [153].

Following this work, the fitting of CNNs onto such local architectures was considered. It appears likely that the difference in dimensionality between 3D biological systems and 2D silicon ICs plays a critical role in limiting the ability to replicate natural systems in efficient and compact artificial architectures. Greater generic and global connectivity is required to provide replacement for the lost dimensionality, resulting in significant resource usage. Future research into 3D chip stacking and new fabrication techniques may alleviate this issue, however until this point it seems likely that the power and resource advantages provided by locally connected architectures will be somewhat mitigated by the complexity of the networks users desire to implement.

In conclusion, this chapter shows that locality and dimensionality both have significant impacts when organising a neural network, and therefore local connectivity should be considered alongside globally connected packet-switch systems when designing new and novel architectures. As seen with FPGA and Application Specific Integrated Circuit (ASIC) solutions, a trade-off between flexibility and efficiency arises. A globally connected system provides maximum flexibility, while a constrained connectivity can yield greater efficiency in terms of power and energy. Since neural networks benefit from

their ability to adapt, the actual post-training flexibility required for neural network implementations is, to date, a question which remains to be answered. It seems likely, however, that some middle-ground between the highly efficient and well constrained naturally occurring systems and the highly flexible globally connected systems may be found to be most suitable for end-user applications and technologies. As a result, it is critical that the communications infrastructure in future neuromorphic systems undergo the same rigorous selection and testing process as that of the neural models themselves. Without innovation in this connectivity, communications between neurons will rapidly become the limiting factor for massive scale neural networks.

Chapter 9

Conclusion

The continued improvement in processor speeds and scales has driven computational technologies for the last century. Gordon Moore famously observed that the number of components applied in Integrated Circuits (ICs) doubled approximately every two years, while Dennards scaling states that the power density of transistors remain constant as their size is reduced. This reliable and incremental trend has allowed new processors and IC technologies to continuously improve in both their computational power and energy consumption. This trend, however, cannot continue indefinitely, with physical scaling limits leading to what is commonly known as the end of Moore's law. The industry is rapidly approaching scales that cannot be reduced and Dennard scaling was observed to break down after 2006, with leakage currents now playing a key role in determining the performance of an IC [4].

With the end of Moore's Law and the fast approaching physical transistor scaling limits, new technologies and design paradigms must be developed to allow the continued improvement of general processor technologies. As this trajectory reaches its long predicted end, designers and engineers must look to other sources of inspiration, such as that of the human brain, to provide future improvements in computing power [6].

Neural networks are ubiquitous as tools for data analysis and information processing. In the early part of the 21st century neural networks found widespread application in many research fields, alongside major integration into new commercial products. These systems typically excel at classification, speech generation and image recognition tasks [5], producing state of the art results that rival custom made and hand tuned algorithms. The scale of these Artificial Neural Networks (ANNs) has grown substantially in recent years, driven largely by increases in processing power and memory capacity. For example, the full SpiNNaker system targets 1 billion neurons and 1 trillion synapses [92], while TrueNorth offers 1 million spiking neurons and 256 million synapses [159]. The increase in network scale has enabled more complex and powerful neural networks to be constructed, and there is a continuing drive to produce even larger networks. As a result, the design of efficient and effective hardware implementations is an important

research topic [34].

Neuromorphic hardware was developed in an effort to realise the efficiency and cognitive potential seen in biological systems. Originally coined by Mead to refer to bio-inspired analogue circuits [6], this class of hardware now includes both analogue and digital circuits designed to offer accelerated computation for either biophysically accurate or computationally efficient neural models and networks.

With potential applications in bio-interfacing and mobile computation, future neuromorphic systems must offer powerful cognitive computing systems using energy efficient and compact solutions. At the other extreme, where High Performance Computing (HPC) implementations of neuromorphic systems are used to perform large scale computation tasks, the efficiency of these systems is equally critical. This is seen in Google's own application of their Tensor Processing Units (TPUs), where they claim to have realised an order of magnitude improvement in performance per watt [83]. The efficiency of the underlying models and hardware used in the production of next generation neuromorphic systems will therefore become a critical metric as the scale and application of these networks increases.

9.1 Analogue Neuromorphic Systems

The original neuromorphic systems were designed to leverage similarities in the non-linear function of biological neurons and transistors. This direct and efficient method of mapping the biological dynamics onto transistor circuits has resulted in a large number of different circuit designs and implementations. Chapter 5 demonstrates one such implementation of a Hodgkin-Huxley model - resulting in an efficient sub-threshold design that uses 29 transistors and 3 capacitors to model the full sodium, potassium and leakage channel dynamics of a biological neuron. This design was fabricated on a $0.35\mu m$ process and used parametrised currents (up to $10\mu A$) and voltages to provide a fully tuneable solution. Simulations demonstrate the implementations ability to produce biophysically plausible Action Potentials (APs) and tonic spiking patterns in response to a supra-threshold stimulus.

9.1.1 Advantages

Power consumption has already been identified as a critical metric by which neuromorphic systems are commonly compared. Analogue implementations often benefit from the non-linear characteristics of transistors, allowing simple circuits to realise relatively complex mathematical relationships. These systems achieve considerably lower power consumption than their digital counterparts while also providing an analogue interface that closely represents that of natures own neural

systems. Analogue circuits are therefore well suited for bio-interfacing applications in particular, where the system is required to interface directly with the biology.

Additionally, these systems often require few components when compared with their digital equivalent, making them a small and effective solution when space is a limited resource. However circuit size is largely dictated by the system requirements meaning that these systems can scale rapidly when specific timing constants or signal magnitudes are required.

9.1.2 Limitations

The decision to incorporate the capacitors into the neuron chip introduced in Chatper 5 was made to ensure that the resulting solution was suitable for integration into bio-medical devices such as pace makers. From the layout shown in figure 5.10 it can be seen that the membrane and channel capacitors use around 90% of the available chip floor-space. This highlights one of the fundamental issues with biophysically accurate analogue neuron ICs. In order to match the relatively slow timing seen in biological neurons, the analogue ICs must either use currents within the deep-sub-threshold region, or large capacitors operating as low-pass filters. These systems become more susceptible to noise as the currents are reduced to allow a reduction in circuit area.

While the chip was based on a published neuron design, implementation of the circuit proved a challenging and time intensive task. This is largely because analogue design is process specific, meaning that implementations of the same design on new technologies or processes require development to ensure that their function and performance meets expectations. This requirement to re-design a solution for each and every fabrication process greatly limits the adoption of analogue designs. Published works in this field can often only describe the underlying design principles or models used to achieve the implementation, leaving individuals to attempt implementation within their own process and pipeline. Small changes in technology can lead to considerable changes in performance when the circuits in question depend on the non-linear and sub-threshold performance characteristics of their internal components.

Adding to this challenge, analogue designs are significantly less common than their digital counterparts. This means that individuals specialising in analogue IC design have become a somewhat limited resource making it impractical for small companies or individuals to support custom analogue solutions.

Finally, process variation was observed to have considerable impact on the timing and response characteristics of the implemented neural model. The falling edge of the AP was shown to vary by more than $1ms$ in a 1,500 point Monte-Carlo simulation, representing a change of as much as 92% of the mean AP timing window. This performance variance is unsurprising when operating with currents of several micro-amps. Small fluctuations within the chip fabric itself, alongside variance in the transistor

dimensions leads to different performance even when produced as part of the same fabrication run, resulting in changes to system timings and even outright model failures. Addressing and mitigating these process variation effects is a time consuming task. External bias values must be used alongside an additional calibration step. This would represent a significant time commitment and is not practical when scaling the neural models to several thousand neurons. Variation as a result of thermal properties would also mean that regular re-calibration was necessary. While this calibration could be automated, this would require the design and test of an additional system which, in and of itself, could suffer from similar fluctuations.

9.1.3 Conclusion

Custom analogue solutions yield some of the smallest and most efficient systems in neuromorphic engineering. It is not unlikely that the analogue nature of the nervous system plays a core role in its inherent efficiency. However, until the nervous systems ability to learn and adapt has been further understood, the issues with reliability and variation common to sub-threshold analogue designs make the production of large scale analogue networks impractical at this current time. The work described in Chapter 5 has shown that an on-chip analogue system can be used to implement a biophysically accurate neural model. For small custom solutions this approach may yield considerable gains over the more generic digital approaches seen in modern neuromorphic systems.

9.2 Digital Neuromorphic Systems

Digital solutions have become commonplace within IC design. With the benefit of well defined libraries that help isolate the design pipeline from the end technology or process, these systems are highly transferable and easily described. Hardware Description Languages (HDLs) allow systems to be tested using Field Programmable Gate Arrays (FPGAs) before implementation on Application Specific Integrated Circuits (ASICs). With the wide adoption of these techniques, the cost and risks associated with digital systems design has been greatly reduced when compared against historic digital systems or analogue solutions.

Digital neuromorphic systems take the established lessons and techniques from Very-Large Scale Integration (VLSI) design and apply them to new novel processors inspired by biological neurons and networks. These systems are often more focused on the computational abilities of a given model, using computationally efficient and somewhat simplified models rather than attempting to replicate nature with biophysically accurate neurons. Where biophysical accuracy is required, however, digital system may still be utilised as shown in Chapter 6. These systems require optimisation or approximation

to make them practical.

9.2.1 Achieving Optimised Biophysical Accuracy

The performance is commonly considered second to model representation accuracy when developing biophysically accurate models, and the Hodgkin-Huxley model is most commonly used as a golden reference against which new models are validated. The model contains multiple Ordinary Differential Equations (ODEs) and internal parameters that each rely heavily on reciprocal and exponential functions. These functions are computationally expensive, and accelerating them at a fundamental level can have a compound effect on the system as a whole. Section 6.3 identified and refined 5 approximations for the exponential function. These approximations were compared against one another for percentage accuracy before their application within a Hodgkin-Huxley model was tested.

From the Hodgkin-Huxley implementation tests it has been shown that a relatively high accuracy about the origin is required for effective tonic spiking replication, while the accuracy across the full supported input range need not be as strict to reliably produce the characteristic AP morphology. The Euler fraction and power series approximations were found to perform poorly without a large number of iterations and this is likely due to the relatively early and rapid climb in error seen as the input magnitude increases.

It was shown that the 2nd order polynomial approximation produces comparable error to that of the optimised 3-line approximation (see Table 6.2). A direct implementation of the polynomial system would require 2 addition and 3 multiply blocks. The 3-line system would require 1 addition and 1 multiply block alongside the logic to select and store the weights used for each of the lines. The relative trade-off between registers and maths blocks depends largely on the underlying technology or hardware used for the final implementation - showing that even the implementation of an effective exponential maths block requires oversight of the end application.

A hybrid model was introduced that uses a unique combination of two different approximations for different input regions. By switching between the approximations, the designer may select different accuracies (and in turn trade-off resource requirements) for target regions. In the basic hybrid model used in Chapter 6, a single line was used to approximate all values above $x = 0.0802$. This line required a single addition operation, representing what can be considered the minimum hardware requirement for any approximation. For values below $x = 0.0802$, the small value approximation $e^x = 1 + x$ was used. This hybrid approximation achieved full representation of the Hodgkin-Huxley model at the expense of a 0.74% deviance in the resting potential. This deviance could be removed fully using a more complicated large input approximation, such as the 3-line piecewise or 2nd order polynomial approximations. This hybrid model's relative success can be attributed to the combination of minimal small value

error and constant maximum error across the full input range.

The relative gains achieved by removing a few maths operations from a neural model can be more easily recognised when considered in system or network scale terms. The basic hybrid model uses 10 fewer multiply operations and 1 less addition when compared with the direct 4th order polynomial model that was the only other non-iterative model to achieve both phasic and tonic spiking. Even assuming that the same exponential block is used for all exponential operations in a single neuron; a simulation of a single neocortical column, which typically contain 10,000 neurons and 30 million connections, would result in 100,000 fewer multiply operations and 10,000 fewer addition operations. This reduction in mathematical operations drastically reduces the scale and power consumption of hardware implementations.

9.2.2 Computationally Efficient Model Acceleration

In computational or artificial neural models, the biophysical dynamics are exchanged for simpler non-linear activation functions. The use of these activation functions allows networks of massive scale to be implemented with considerable ease when compared against their biophysically accurate counterparts. The exponential function still sees considerable use within these models, however when optimising for hardware implementations the emphasis is on non-linear form rather than absolute value. This means that approximations that produce the correct shape can perform with comparable results to that of a fully implemented activation function.

The logistic sigmoid, tanh, gaussian and softplus activation functions all use natural exponentials in their computation. Chapter 7 shows that this natural exponential may be directly replaced with a power of two. The effect of this replacement is seen in the first order derivatives, where an additional $ln(2)$ factor appears, as shown in Table 7.2. This scaling of the derivative values, however, may be absorbed into the learning rate constant. By replacing the exponential powers with powers of two the calculations may be performed as simple fixed-point bit-shift operations. A conversion technique to migrate models between the full exponential and base-2 models was demonstrated, resulting in identical performance for all networks tested. This conversion makes the base-2 models more applicable to the research community as fully trained systems need not undergo re-training to migrate between the two representations.

The base-2 and approximation models were compared against the exponential sigmoid model, using a mixture of generated and established datasets. These comparisons show that model selection has a small impact on the systems performance, however this impact lacks any general trend when comparing the sigmoid and base-2 models. This suggests that neither model is better than the other in performance as a general purpose classification system.

A double-precision floating point hardware accelerator was built using the bitwise

approximation activation function allowing direct comparison with modern day processing solutions. When compared against modern processor technologies, the hardware bitwise accelerator operation achieved a throughput of 87.7MHz at a rate of 1 clock cycle per calculation. This beats the throughput of an exponential operation on the next best processor by 0.7MHz, which additionally required 40 clock cycles at 3.5GHz.

The results of this study show that new base-2 models may be used in place of exponential models to achieve similar, if not identical, performance at a reduced computational cost. This approach of designing the models from the ground up with hardware optimisation in mind could be applied to many other aspects of neuromorphic engineering. Models such as the base-2 sigmoid proposed in this work will play a critical role in the adoption of ANNs on both low-power constrained mobile devices and high-power large scale server implementations. The design of hardware accelerators that complement modern processor techniques provides a good intermediate solution that allows generic processors to realise power and speed improvements without considerable architecture redesign.

9.3 Designing for Network Connectivity

The communications infrastructure used to model synapses and connect neurons together is another major challenge on large scale neuromorphic systems. As neurons are accelerated, the communications bottleneck will rapidly become the constraining factor in network scale and speed. Most modern neuromorphic system utilise packet switching networks to achieve the appearance of all-to-all connectivity while saving on hardware resources. These systems use lessons from large scale computer networks and communications networks to achieve a flexible and relatively low power solution. This represents a considerable change from the structure and communications seen in biological neural systems. It is therefore interesting to consider how the principles of biological synapses and connectivity may be implemented in hardware.

This difference in connectivity is best highlighted using Rent's rule. Apply Rent's rule to the Human Nervous System (HNS) and the *C. Elegans* and an average Rent coefficient of 0.77 is found. In contrast this coefficient is 1.0 for all-to-all connected networks. This suggests that there is a sparsity in connectivity that may be utilised in the design of effective hardware neuromorphic systems. A novel locally connected grid architecture was introduced in Section 8.2.2. Using principles from FPGA design, this architecture is built from simple sub-elements with support for reconfigurable operation and connectivity. This configurability was demonstrated using the *C. Elegans* locomotive model, and was shown to produce reliable and correct response to stimulus when compared against previously published works. A 3D model of the *C. Elegans* was built and connected to the architecture allowing visualisation of the animals locomotive response to different stimuli.

The *C. Elegans* model achieved a constant clock speed of 82.13MHz on the locally connected grid architecture, regardless of the number of model segments simulated. Compared against the hybrid column architecture by Bailey, this represents an improvement of 3.5× to 17.5× in maximum clock-speed showing that systems which reflect the underlying models connectivity can yield considerable improvements over generic fully connected architectures.

9.3.1 Connectivity Limitations in Artificial Systems

High degrees of locality are also seen in artificial systems, such as Convolutional Neural Networks (CNNs) where the convolutional layers only access a local neighbourhood of values from the previous layer. This locality is not unsurprising since CNNs are heavily inspired by biological systems, where long range connections have a high cost in space, delay and noise. Despite this apparent locality, it was identified that these systems do not map effectively onto locally connected architectures. This is largely due to the differences in the dimensionality of silicon hardware implementations and the CNNs themselves. CNNs effectively form 3-dimensional systems, with each convolutional layer forming an additional image layer stacked along the 3rd dimension. Standard IC solutions, on the other hand, are said to be 2.5D in recognition of the fact that they are formed from a discrete number of 2D layers. This loss in dimensionality means that the stacking of the convolutional layers must be unravelled onto the 2D structure. Performing this task heavily increases the utilisation and requirement for long range connections. This limitation makes it difficult to match the performance and speed seen in biological systems where there is an inherent 3-dimensional structure.

This dimensionality constraint is not unique to the neuromorphic field and future technologies may help close this gap as new fabrication techniques are developed. Until this point, it is likely that locally connected architectures will be somewhat limited in their application, with global systems effectively replacing the lost dimension with long-range connections that allows the structures to be unrolled onto a 2D system.

9.4 Future Directions and Challenges

There are many more challenges that must be overcome for neuromorphic computing to realise the efficiency and computational potential that is seen in nature. This multi-disciplinary field must still answer significant questions regarding the nature of learning and consciousness, as well as identifying the critical elements of the nervous system that contribute to its processing functions. From an engineering perspective, careful selection and design of neuromorphic architectures stands to accelerate this research, directly improving the energy efficiency and increasing the scales at which experiments may be performed. This work has demonstrated a number of design improvements to

that end, however further optimisation and design exploration is still possible.

The bitwise sigmoid hardware accelerator design in Chapter 7 realised a significant efficiency improvement over standard Central Processing Unit (CPU) implementations. The implementation, however, used 68 Digital Signal Processing (DSP) elements. These DSPs were predominantly required to implement a basic 58-bit multiply performed as part of the Newton-Raphson reciprocal block. This multiply operation represents the slowest signal path in the system and is therefore also the limiting factor on maximum clock speed. Techniques to perform fast and efficient pipelined fixed-point multiply operations are well established within the engineering community and it would therefore be interesting to investigate how the custom instruction may be further accelerated and reduced in scale by redesigning the multiply implementation. Improvements to this multiply operation stands to accelerate the maximum clocks speed of the entire implementation, further advancing the gains achieved through hardware acceleration. While this custom instruction was designed for FPGA implementation, it would also be interesting to test the design on an ASIC device where additional speed and power savings may be realised.

The EPSRC CResPace project, discussed in Chapter 5, continues to investigate the development of adaptive bio-electronics for chronic cardiorespiratory disease. The MD-Neuron chip was demonstrably functional in large static magnetic fields, meaning that the design can safely approach non-operational Magnetic Resonance Imaging (MRI) scanners. Further tests must now be performed within dynamic field conditions to ensure that the device remains operational during MRI procedures. The dynamic fields will induce currents within the IC, and it is therefore important to assess how these additional currents impact the operation of the device. Arranging these tests represents a considerable challenge for the CResPace project, as any experiments involving dynamic (imaging) magnetic fields require a qualified radiographer and access to a functional medical MRI machine. In the time taken to arrange these experiments further design work may also be performed, building on the results in Chapter 5. A functional pacemaker with a novel neural-interface must be designed, using the neurons demonstrated in the MD-Neuron IC to form a Central Pattern Generator (CPG) that controls the heart rate. This design work may progress alongside the dynamic field tests, allowing the CResPace team to test the latest design when an MRI machine becomes available. Even with these later designs, some further adjustments should be expected post-MRI test to help ensure the product is immune to any negative effects of the dynamic field. It is hoped that the CResPace experiments will help inform research on the application of neural-interfacing circuitry for next-generation prosthetics and pacemakers.

Finally, Chapter 8 identified a fundamental difference in dimensionality between that of neuromorphic architectures and biological neural networks. This dimensionality difference is unlikely to be addressed while ICs remain constrained to a fixed number of 2D layers. Continued developments in new fabrication materials and technologies may, one day, provide solutions for this problem, with attempts such as chip-stacking already

showing promising results in improving the speed and efficiency of hardware solutions [160]. Future 3-dimensional IC designs may achieve the efficiencies and computational powers seen in nature, leading to a revolution in processing and artificial intelligence systems.

9.5 Closing Thoughts

While future technologies promise faster and larger neural networks, it is possible to achieve improved efficiency in today's systems through the careful and deliberate design of a networks internal functions. This principle has been shown using the base-2 sigmoid function, where a clock-cycles per calculation improvement of 97.5% was achieved over the industry standard model. The efficiency and effectiveness of these models grows ever more important as these systems find application within day to day life. Neural models stand to improve bio-electronics, computer science, control engineering, finance and many other important fields. Improvements in these models that allow the network scales to increase or chip area and power to decrease will lead to more powerful and compact systems that provide exciting and novel solutions to previously challenging problems.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, 1965.

[2] ——, "Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 36–37, sep 2006.

[3] R. Dennard, F. Gaensslen, Hwa-Nien Yu, V. Rideout, E. Bassous, and A. Leblanc, "Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions," *Proceedings of the IEEE*, vol. 87, no. 4, pp. 668–678, apr 1999.

[4] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[5] W. Rawat and Z. Wang, "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review," *Neural Computation*, vol. 29, no. 9, pp. 2352–2449, sep 2017.

[6] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990.

[7] N. S. Ward, "Neural plasticity and recovery of function," *Progress in Brain Research*, vol. 150, pp. 527–535, jan 2005.

[8] P. Bach-y Rita, "Brain plasticity as a basis for recovery of function in humans," *Neuropsychologia*, vol. 28, no. 6, pp. 547–554, jan 1990.

[9] P. L. Williams, M. Berry, L. H. Bannister, and S. M. Standring, "Nervous System," in *Gray's Anatomy*, 38th ed., 1995, pp. 901–1397.

[10] F. A. Azevedo, L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, W. J. Filho, R. Lent, and S. Herculano-Houzel, "Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain," *The Journal of Comparative Neurology*, vol. 513, no. 5, pp. 532–541, apr

2009.

[11] B. Fischl and A. M. Dale, "Measuring the thickness of the human cerebral cortex from magnetic resonance images," *Proceedings of the National Academy of Sciences*, vol. 97, no. 20, pp. 11 050–11 055, sep 2000.

[12] H. Haken, *Brain Dynamics*, ser. Springer Series in Synergetics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[13] E. Izhikevich, *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*, 2007.

[14] G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S. Liu, P. Dudek, P. Hafliger, S. Renaud, J. Schemmel, G. Cauwenberghs, J. Arthur, K. Hynna, F. Folowosele, S. Saighi, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang, and K. Boahen, "Neuromorphic Silicon Neuron Circuits," *Frontiers in Neuroscience*, vol. 5, no. May, pp. 1–23, 2011.

[15] J. A. Bailey, "Towards the neurocomputer: An investigation of VHDL Neuron Models," PhD Thesis, University of Southampton, 2010.

[16] W. A. Catterall, I. M. Raman, H. P. C. Robinson, T. J. Sejnowski, and O. Paulsen, "The Hodgkin-Huxley Heritage: From Channels to Circuits," *Journal of Neuroscience*, vol. 32, no. 41, pp. 14 064–14 073, oct 2012.

[17] A. L. Hodgkin, A. F. Huxley, and B. Katz, "Measurement of current-voltage relations in the membrane of the giant axon of Loligo," *Journal of Physiology*, vol. 116, pp. 424–448, 1952.

[18] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500–544, jan 1952.

[19] ——, "Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo," *Journal of Physiology*, vol. 116, pp. 449–472, 1952.

[20] E. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, nov 2003.

[21] ——, "Which Model to Use for Cortical Spiking Neurons?" *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, sep 2004.

[22] R. D. Hayes, J. H. Byrne, and D. A. Baxter, "Neurosimulation: Tools and Resources," in *The Handbook of Brain Theory and Neural Networks, 2nd edition*, 2002, pp. 776–780.

[23] M. L. Hines and N. T. Carnevale, "NEURON Simulation Environment," in *The*

Handbook of Brain Theory and Neural Networks, 2nd edition, 2002, pp. 769–773.

[24] M. Migliore, C. Cannia, W. W. Lytton, H. Markram, and M. L. Hines, "Parallel network simulations with NEURON," *Journal of Computational Neuroscience*, vol. 21, no. 2, pp. 119–129, oct 2006.

[25] J. W. Moore and A. E. Stuart, "Neurons In Action: Computer Simulations with NeuroLab," *The Journal of Undergraduate Neuroscience Education*, vol. 2, no. 2, p. 2, 2004.

[26] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris, M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. El Boustani, and A. Destexhe, "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neuroscience*, vol. 23, no. 3, pp. 349–398, dec 2007.

[27] R. A. Tikidji-Hamburyan, V. Narayana, Z. Bozkus, and T. A. El-Ghazawi, "Software for Brain Network Simulations: A Comparative Study," *Frontiers in Neuroinformatics*, vol. 11, no. July, pp. 1–16, jul 2017.

[28] M. Diesmann and M.-O. Gewaltig, "NEST: An environment for neural systems simulations," *Forschung und wisschenschaftliches Rechnen, Beitrage zum Heinz-Billing-Preis*, vol. 58, pp. 43–70, 2001.

[29] A. Morrison, C. Mehring, T. Geisel, A. Aertsen, and M. Diesmann, "Advancing the Boundaries of High-Connectivity Network Simulation with Distributed Computing," *Neural Computation*, vol. 17, no. 8, pp. 1776–1801, aug 2005.

[30] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "PyNN: a common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, vol. 2, 2008.

[31] A. Mordvintsev, C. Olah, and M. Tyka, "Inceptionism: Going Deeper into Neural Networks," 2015.

[32] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, oct 2017.

[33] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A Survey of Neuromorphic Computing and Neural Networks in Hardware," *CoRR*, pp. 1–88, may 2017.

[34] D. Monroe, "Neuromorphic computing gets ready for the (really) big time," *Communications of the ACM*, vol. 57, no. 6, pp. 13–15, 2014.

[35] P. D. Wasserman, *Neural Computing: Theory and Practice*. Coriolis Group C/O Publishing Resources Inc, 1989.

[36] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, dec 1943.

[37] D. O. Hebb, "The Organization of Behavior," 1949.

[38] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[39] L. Abbott, "Lapicque's introduction of the integrate-and-fire model neuron (1907)," *Brain Research Bulletin*, vol. 50, no. 5-6, pp. 303–304, nov 1999.

[40] L. Lapicque, "Recherches quantitatives sur I'excitation electrique des nerfs traitee comme une polarization," *J. Physiol. Paris*, vol. 9, pp. 620–635, 1907.

[41] C. Bishop, *Pattern recognition and machine learning*. New York, NY: Springer, 2006.

[42] N. Leibowitz, B. Baum, G. Enden, and A. Karniel, "The exponential learning equation as a function of successful trials results in sigmoid performance," *Journal of Mathematical Psychology*, vol. 54, no. 3, pp. 338–340, jun 2010.

[43] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade*, G. B. Orr and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50.

[44] B. Karlik and A. Vehbi, "Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks," *International Journal of Artificial Intelligence And Expert Systems (IJAE)*, vol. 1, no. 4, pp. 111–122, 2010.

[45] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for Activation Functions," pp. 1–13, oct 2017.

[46] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, dec 2017.

[47] J. Pennington, S. S. Schoenholz, and S. Ganguli, "Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice," no. Nips, pp. 1–11, nov 2017.

[48] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann

Machines," *Proceedings of the 27th International Conference on Machine Learning*, no. 3, pp. 807–814, 2010.

[49] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," *Proceedings of the 30 th International Conference on Machine Learning*, vol. 28, p. 6, 2013.

[50] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1972.

[51] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4, pp. 303–314, dec 1989.

[52] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[53] Y. Bengio, "Learning Deep Architectures for AI," *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[54] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, jan 2015.

[55] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur, "Recurrent Neural Network based Language Model," *Interspeech*, no. September, pp. 1045–1048, 2010.

[56] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur, "Extensions of recurrent neural network language model," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, may 2011, pp. 5528–5531.

[57] I. Sutskever, J. Martens, and G. Hinton, "Generating Text with Recurrent Neural Networks," in *28th International Conference on Machine Learning*, 2011.

[58] S. Liu, N. Yang, M. Li, and M. Zhou, "A Recursive Recurrent Neural Network for Statistical Machine Translation," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2014, pp. 1491–1500.

[59] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," sep 2014.

[60] A. Graves and N. Jaitly, "Towards End-To-End Speech Recognition with Recurrent Neural Networks," in *Proceedings of the 31st International Conference on Machine Learning*, vol. 32, no. 1, Beijing, 2014, pp. 1764–1772.

[61] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, no. 6. IEEE, may 2013, pp. 6645–6649.

[62] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, jul 2005.

[63] D. Eck and J. Schmidhuber, "Learning the Long-Term Structure of the Blues," in *Proceedings of Int. Conf. on Artificial Neural Networks—ICANN 2002*, 2002, pp. 284–289.

[64] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, vol. 1, no. 4, pp. 339–356, jan 1988.

[65] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, nov 1997.

[66] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber, "A Novel Connectionist System for Unconstrained Handwriting Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 855–868, may 2009.

[67] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, apr 1982.

[68] W. Little, "The existence of persistent states in the brain," *Mathematical Biosciences*, vol. 19, no. 1-2, pp. 101–120, feb 1974.

[69] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons." *Proceedings of the National Academy of Sciences*, vol. 81, no. 10, pp. 3088–3092, may 1984.

[70] G. E. Hinton and T. J. Sejnowski, "Learning and Relearning in Boltzmann Machines," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, C. Rumelhart, David E. and McClelland, James L. and PDP Research Group, Ed. Cambridge, MA, USA: MIT Press, 1986, pp. 282—-317.

[71] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex." *The Journal of physiology*, vol. 195, no. 1, pp. 215–43, mar 1968.

[72] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object Recognition with Gradient-Based Learning," 1999, no. 0, pp. 319–345.

[73] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief

networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*. New York, New York, USA: ACM Press, 2009, pp. 1–8.

[74] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1.* Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.

[75] D. Parker, "Learning Logic," in *Invention Report S81-64, File 1*, Stanford University. Stanford CA.: Office of Technology Licensing, 1982.

[76] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1974, vol. PhD thesis, no. January 1974.

[77] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly of Applied Mathematics*, vol. 2, no. 2, pp. 164–168, jul 1944.

[78] D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[79] H. Yu and B. Wilamowski, "Levenberg–Marquardt Training," in *Industrial Electronics Handbook*, 2nd ed., feb 2011, pp. 1–16.

[80] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4. IEEE, 1995, pp. 1942–1948.

[81] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*. IEEE, 1998, pp. 69–73.

[82] Y. Zhang, S. Wang, G. Ji, Y. Zhang, S. Wang, and G. Ji, "A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications," *Mathematical Problems in Engineering*, vol. 2015, pp. 1–38, 2015.

[83] N. P. Jouppi, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, C. Young, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, N. Patil, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Patterson, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, G. Agrawal, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross,

A. Salek, R. Bajwa, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, S. Bates, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D. H. Yoon, S. Bhatia, and N. Boden, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 1–12, jun 2017.

[84] R. Silver, K. Boahen, S. Grillner, N. Kopell, and K. L. Olsen, "Neurotech for Neuroscience: Unifying Concepts, Organizing Principles, and Emerging Tools," *Journal of Neuroscience*, vol. 27, no. 44, pp. 11 807–11 819, oct 2007.

[85] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing Machines," *CoRR*, pp. 1–26, oct 2014.

[86] S. Furber, S. Temple, and A. Brown, "On-chip and inter-chip networks for modeling large-scale neural systems," in *2006 IEEE International Symposium on Circuits and Systems*. IEEE, 2006, p. 4.

[87] S. B. Furber, S. Temple, and A. Brown, "High-performance computing for systems of spiking neurons," in *Adaptation in Artificial and Biological Systems*, vol. 2, Bristol, 2006, pp. 29–36.

[88] E. Stromatias, D. Neil, F. Galluppi, M. Pfeiffer, S.-c. Liu, and S. Furber, "Scalable energy-efficient, low-latency implementations of trained spiking Deep Belief Networks on SpiNNaker," in *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, jul 2015, pp. 1–8.

[89] J. Bainbridge and S. Furber, "Chain: a delay-insensitive chip area interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16–23, sep 2002.

[90] E. Painkras, L. A. Plana, J. Garside, S. Temple, S. Davidson, J. Pepper, D. Clark, C. Patterson, and S. Furber, "SpiNNaker: A multi-core System-on-Chip for massively-parallel neural net simulation," in *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*, vol. 4. IEEE, sep 2012, pp. 1–4.

[91] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the SpiNNaker System Architecture," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454–2467, dec 2013.

[92] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, "SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1943–1953, aug 2013.

[93] E. Stromatias, F. Galluppi, C. Patterson, and S. Furber, "Power analysis of large-scale, real-time neural networks on SpiNNaker," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*. IEEE, aug 2013, pp. 1–8.

[94] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-j. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, oct 2015.

[95] R. Preissl, T. M. Wong, P. Datta, M. Flickner, R. Singh, S. K. Esser, W. P. Risk, H. D. Simon, and D. S. Modha, "Compass: A scalable simulator for an architecture for cognitive computing," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, nov 2012, pp. 1–11.

[96] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman, A. Amir, D. B.-D. Rubin, F. Akopyan, E. McQuinn, W. P. Risk, and D. S. Modha, "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*. IEEE, aug 2013, pp. 1–10.

[97] A. S. Cassidy, R. Alvarez-Icaza, F. Akopyan, J. Sawada, J. V. Arthur, P. A. Merolla, P. Datta, M. G. Tallada, B. Taba, A. Andreopoulos, A. Amir, S. K. Esser, J. Kusnitz, R. Appuswamy, C. Haymes, B. Brezzo, R. Moussalli, R. Bellofatto, C. Baks, M. Mastro, K. Schleupen, C. E. Cox, K. Inoue, S. Millman, N. Imam, E. Mcquinn, Y. Y. Nakamura, I. Vo, C. Guok, D. Nguyen, S. Lekuch, S. Asaad, D. Friedman, B. L. Jackson, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "Real-Time Scalable Cortical Computing at 46 Giga-Synaptic OPS/Watt with ˜100x Speedup in Time-to-Solution and ˜100,000x Reduction in Energy-to-Solution," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, vol. 2015-Janua, no. January. IEEE, nov 2014, pp. 27–38.

[98] B. V. Benjamin, Peiran Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, may 2014.

[99] P. Merolla, J. Arthur, R. Alvarez, J.-M. Bussat, and K. Boahen, "A Multicast Tree Router for Multichip Neuromorphic Systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 3, pp. 820–833, mar 2014.

[100] S. Choudhary, S. Sloan, S. Fok, A. Neckar, E. Trautmann, P. Gao, T. Stewart, C. Eliasmith, and K. Boahen, "Silicon Neurons That Compute," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012, vol. 7552 LNCS, no. PART 1, pp. 121–128.

[101] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and Evaluation of the First Tensor Processing Unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, may 2018.

[102] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, jan 2018.

[103] C.-K. Lin, A. Wild, G. N. Chinya, Y. Cao, M. Davies, D. M. Lavery, and H. Wang, "Programming Spiking Neural Networks on Intel's Loihi," *Computer*, vol. 51, no. 3, pp. 52–61, mar 2018.

[104] J. Hsu, "IBM's new brain [News]," *IEEE Spectrum*, vol. 51, no. 10, pp. 17–19, oct 2014.

[105] J. A. Walker, M. A. Trefzer, S. J. Bale, and A. M. Tyrrell, "PAnDA: A Reconfigurable Architecture that Adapts to Physical Substrate Variations," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1584–1596, aug 2013.

[106] C. Bartolozzi and G. Indiveri, "Synaptic Dynamics in Analog VLSI," *Neural Computation*, vol. 19, no. 10, pp. 2581–2603, oct 2007.

[107] P. Merolla and K. Boahen, "A Recurrent Model of Orientation Maps with Simple and Complex Cells," *Advances in Neural Information Processing Systems*, vol. 16, no. December, pp. 995–1002, 2004.

[108] J. Hasler, A. Natarajan, and S. Kim, "Enabling Energy-Efficient Physical Computing through Analog Abstraction and IP Reuse," *Journal of Low Power Electronics and Applications*, vol. 8, no. 4, p. 47, nov 2018.

[109] R. B. Thompson and E. R. McVeigh, "Cardiorespiratory-resolved magnetic resonance imaging: Measuring respiratory modulation of cardiac function," *Magnetic Resonance in Medicine*, vol. 56, no. 6, pp. 1301–1310, dec 2006.

[110] M. Mahowald and R. Douglas, "A silicon neuron," *Nature*, vol. 354, no. 6354, pp. 515–518, dec 1991.

[111] C. Mead, *Analog VLSI and Neural System*. Addison Wesley Publishing Company, 1989.

[112] L. Hebrard, D. V. Nguyen, D. Vogel, J.-B. Schell, C. Po, N. Dumas, W. Uhring, and J. Pascal, "On the influence of strong magnetic field on MOS transistors," in *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, dec 2016, pp. 564–567.

[113] L. Euler, *Institutionum calculi integralis*, ser. Institutionum calculi integralis. imp. Acad. imp. Saènt., 1768, no. v. 1.

[114] S. Gomar, M. Mirhassani, and M. Ahmadi, "Precise digital implementations of hyperbolic tanh and sigmoid function," in *2016 50th Asilomar Conference on Signals, Systems and Computers*, no. 5. IEEE, nov 2016, pp. 1586–1589.

[115] K. Basterretxea, J. Tarela, and I. del Campo, "Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons," *IEE Proceedings - Circuits, Devices and Systems*, vol. 151, no. 1, p. 18, 2004.

[116] R. R. Osorio, "Pipelined FPGA implementation of numerical integration of the Hodgkin-Huxley model," *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, vol. 2016-Novem, no. 3, pp. 202–206, 2016.

[117] Altera, *Floating Point Exponent (ALTFP _ EXP) Megafunction User Guide*, 1st ed., 2008, no. December.

[118] M. Schmidt, R. Bakker, K. Shen, G. Bezgin, M. Diesmann, and S. J. van Albada, "A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas," *PLOS Computational Biology*, vol. 14, no. 10, p. e1006359, oct 2018.

[119] H. Kwan, "Simple sigmoid-like activation function suitable for digital hardware implementation," *Electronics Letters*, vol. 28, no. 15, p. 1379, 1992.

[120] H. Faiedh, Z. Gafsi, and K. Besbes, "Digital hardware implementation of sigmoid function and its derivative for artificial neural networks," in *ICM 2001 Proceedings. The 13th International Conference on Microelectronics.*, vol. 2001-Janua. IEEE, 2001, pp. 189–192.

[121] A. Gupta and N. Bhat, "Asymmetric cross-coupled differential pair configuration to realize neuron activation function and its derivative," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, no. 1, pp. 10–13, jan 2005.

[122] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, "Efficient hardware implementation of the hyperbolic tangent sigmoid function," in *2009 IEEE International Symposium on Circuits and Systems*. IEEE, may 2009, pp. 2117–2120.

[123] C.-H. Tsai, Y.-T. Chih, W. H. Wong, and C.-Y. Lee, "A Hardware-Efficient Sigmoid Function With Adjustable Precision for a Neural Network System," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 11, pp. 1073–1077, nov 2015.

[124] *IEEE Std 754^{TM}-2008 (Revision of IEEE Std 754-1985), IEEE Standard for*

Floating-Point Arithmetic, 2008, no. August.

[125] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[126] Y. LeCun, L. D. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. a. Muller, E. Sackinger, P. Simard, and V. Vapnik, "Learning algorithms for classification: A comparison on handwritten digit recognition," in *Neural networks: the statistical mechanics perspective*. World Scientific, 1995, pp. 261–276.

[127] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, jun 2012, pp. 3642–3649.

[128] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, "Regularization of Neural Network using DropConnect," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 40, no. 4, apr 2013, pp. 863–875.

[129] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[130] M. LaPedus, "Intel-Altera deal to shake up foundry landscape," *Chip Design Magazine*, 2013. [Online]. Available: http://chipdesignmag.com/display.php?articleId=5215

[131] M. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proceedings - Computers and Digital Techniques*, vol. 150, no. 6, p. 403, 2003.

[132] D. B. Chklovskii, T. Schikorski, and C. F. Stevens, "Wiring optimization in cortical circuits." *Neuron*, vol. 34, no. 3, pp. 341–7, apr 2002.

[133] W. R. Crowther, F. E. Heart, A. A. McKenzie, J. M. McQuillan, and D. C. Walden, "Issues in packet switching network design," in *Proceedings of the May 19-22, 1975, national computer conference and exposition on - AFIPS '75*. New York, New York, USA: ACM Press, 1975, p. 161.

[134] J. V. Arthur, P. a. Merolla, F. Akopyan, R. Alvarez, A. Cassidy, S. Chandra, S. K. Esser, N. Imam, W. Risk, D. B. D. Rubin, R. Manohar, and D. S. Modha, "Building block of a programmable neuromorphic substrate: A digital neurosynaptic core," in *The 2012 International Joint Conference on Neural Networks (IJCNN)*. IEEE, jun 2012, pp. 1–8.

[135] K. H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa, and W. Lu, "A functional hybrid memristor crossbar-array/CMOS system for data storage and neuromorphic applications," *Nano Letters*, vol. 12, no. 1, pp. 389–395, 2012.

[136] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner, "The Structure of the Nervous System of the Nematode Caenorhabditis elegans," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 314, no. 1165, pp. 1–340, nov 1986.

[137] E. T. Claverol, "An event-driven approach to biologically realistic simulation of neural aggregates," Ph.D. dissertation, University of Southampton, 2000.

[138] K. Fujita and Y. Kashimori, "Neural Mechanism of Corticofugal Modulation of Tuning Property in Frequency Domain of Bat's Auditory System," *Neural Processing Letters*, vol. 43, no. 2, pp. 537–551, apr 2016.

[139] D. Purves, G. J. Augustine, and D. Fitzpatrick, "Autonomic Regulation of the Bladder," in *Neuroscience. 2nd Edition*, P. Et al, Ed. Sunderland, MA: Sinauer Associates, 2001, ch. 20, pp. 493–496.

[140] M. Y. Lanzerotti, G. Fiorenza, and R. A. Rand, "Microminiature packaging and integrated circuitry: The work of E. F. Rent, with an application to on-chip interconnection requirements," *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 777–803, jul 2005.

[141] B. Landman and R. Russo, "On a Pin Versus Block Relationship For Partitions of Logic Graphs," *IEEE Transactions on Computers*, vol. C-20, no. 12, pp. 1469–1479, dec 1971.

[142] P. Christie and D. Stroobandt, "The interpretation and application of Rent's rule," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 6, pp. 639–648, dec 2000.

[143] D. L. Greenfield, "Communication Locality in Computation: Software, Chip Multiprocessors and Brains," Ph.D. dissertation, University of Cambridge, 2010.

[144] D. S. Bassett, D. L. Greenfield, A. Meyer-Lindenberg, D. R. Weinberger, S. W. Moore, and E. T. Bullmore, "Efficient Physical Embedding of Topologically Complex Information Processing Networks in Brains and Computer Circuits," *PLoS Computational Biology*, vol. 6, no. 4, p. e1000748, apr 2010.

[145] P. Wilson, B. Metcalfe, J. Graham-Harper-Cater, and J. A. Bailey, "A reconfigurable architecture for real-time digital simulation of neurons," in *2017 Intelligent Systems Conference (IntelliSys)*, no. September. IEEE, sep 2017, pp. 66–75.

[146] E. Claverol, R. Cannon, J. Chad, and A. Brown, "Event based neuron models for biological simulation . A model of the locomotion circuitry of the nematode C . Elegans ." *World Scientific Engineering Society Press*, 1999.

[147] J. A. Bailey, P. R. Wilson, A. D. Brown, and J. Chad, "Behavioral simulation of biological neuron systems using VHDL and VHDL-AMS," in *2007 IEEE International Behavioral Modeling and Simulation Workshop*. IEEE, sep 2007, pp. 153–158.

[148] J. A. Bailey, P. R. Wilson, A. D. Brown, and J. E. Chad, "Behavioural simulation and synthesis of biological neuron systems using VHDL," *BMAS 2008 - Proceedings of the 2008 IEEE International Behavioral Modeling and Simulation Workshop*, pp. 7–12, 2008.

[149] J. Bailey, R. Wilcock, P. Wilson, and J. Chad, "Behavioral simulation and synthesis of biological neuron systems using synthesizable VHDL," *Neurocomputing*, vol. 74, no. 14-15, pp. 2392–2406, jul 2011.

[150] J. Karbowski, G. Schindelman, C. J. Cronin, A. Seah, and P. W. Sternberg, "Systems level circuit model of C. elegans undulatory locomotion: mathematical modeling and molecular genetics," *Journal of Computational Neuroscience*, vol. 24, no. 3, pp. 253–276, jun 2008.

[151] R. Mailler, J. Avery, J. Graves, and N. Willy, "A Biologically Accurate 3D Model of the Locomotion of Caenorhabditis Elegans," in *2010 International Conference on Biosciences*, vol. 2, no. 3-4. IEEE, mar 2010, pp. 84–90.

[152] E. Niebur and P. Erdös, "Theory of the locomotion of nematodes," *Biophysical Journal*, vol. 60, no. 5, pp. 1132–1146, nov 1991.

[153] J. H. Boyle, S. Berri, and N. Cohen, "Gait Modulation in C. elegans: An Integrated Neuromechanical Model," *Frontiers in Computational Neuroscience*, vol. 6, no. March, pp. 1–15, 2012.

[154] Y. Jin, E. Jorgensen, E. Hartwieg, and H. R. Horvitz, "The Caenorhabditis elegans Gene unc-25 Encodes Glutamic Acid Decarboxylase and Is Required for Synaptic Transmission But Not Synaptic Development," *The Journal of Neuroscience*, vol. 19, no. 2, pp. 539–548, jan 1999.

[155] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*. New York, New York, USA: ACM Press, 2015, pp. 161–170.

[156] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *Proceedings of the 35th International Conference on Computer-Aided Design -*

ICCAD '16, no. August. New York, New York, USA: ACM Press, 2016, pp. 1–8.

[157] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, aug 2017.

[158] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, S. Zheng, T. Lu, J. Gu, L. Liu, and S. Wei, "A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, apr 2018.

[159] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, aug 2014.

[160] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE, aug 2011, pp. 1–24.

Appendix A

Defining Protein Switching Probability Equations

Let it be assumed that a number of proteins exist, $[x]$, that can take one of two different states. The rates of moving between these states may be defined as α_x and β_x, such that the following rate equation may be written:

$$\underset{\text{Deactivated}}{[x]} \underset{\beta_x}{\overset{\alpha_x}{\rightleftharpoons}} \underset{\text{Activated}}{[x^*]} \tag{A.1}$$

Assuming that these switching rates are constant, a differential form may be defined from this rate equation as follows:

$$\frac{d\,[x^*]}{dt} = -\underbrace{\beta_x\,[x^*]}_{\text{Becoming De-active}} + \overbrace{\alpha_x\,[x]}^{\text{Becoming Active}} \tag{A.2}$$

If this differential form is divided by the total amount of protein (that is $[x] + [x^*]$), this yields:

$$\frac{d}{dt} \cdot \frac{[x^*]}{[x] + [x^*]} = \alpha_x \frac{[x]}{[x] + [x^*]} - \beta_x \frac{[x^*]}{[x] + [x^*]} \tag{A.3}$$

Let x be defined as proportion of active proteins, such that:

$$x = \frac{[x^*]}{[x] + [x^*]} \tag{A.4}$$

This value will sit within the range $x \in [1, 0]$ as it is a proportion. In a similar way the proportion of deactivated proteins, $(1 - x)$, may be defined as follows:

$$1 - x = \frac{[x]}{[x] + [x^*]} \tag{A.5}$$

Using these proportion definitions in Equation A.3, the rate at which the protein activates may be found:

$$\frac{dx}{dt} = \overbrace{\alpha_x (1 - x)}^{\text{Becoming Active}} - \underbrace{\beta_x x}_{\text{Becoming De-active}} \tag{A.6}$$

where x is the proportion of active proteins in a sample, while α_x is the activation rate and β_x is the inactivation rate of said protein.

Appendix B

Numerical Solution for the Hodgkin Huxley Model

For ease of reference the Hodgkin-Huxley model is restated below in Equation set B.1, B.2 and B.3. In this model, V is the membrane potential; C_M is the membrane capacitance; I is the injected current; E_K, E_{Na} and E_L are the potassium, sodium and leakage Nernst potentials; n and m are the potassium and sodium activation gating variables; h is the sodium inactivation gating variable; and $\bar{g}_K$, $\bar{g}_{Na}$ and $\bar{g}_L$ are the maximum potassium, sodium and leakage conductances. Equation B.1 is termed the Hodgkin-Huxley equation; Equation set B.2 are the gating variable equations; and Equation set B.3 are the rate equations.

$$C_M \frac{dV}{dt} = I - \overbrace{\bar{g}_K\, n^4(V - E_K)}^{I_k} - \overbrace{\bar{g}_{Na}\, m^3 h(V - E_{Na})}^{I_{Na}} - \overbrace{\bar{g}_L(V - E_L)}^{I_L} \tag{B.1}$$

$$\begin{aligned} \frac{dn}{dt} &= \alpha_n(V) \cdot (1 - n) - \beta_n(V) \cdot n \\ \frac{dm}{dt} &= \alpha_m(V) \cdot (1 - m) - \beta_m(V) \cdot m \\ \frac{dh}{dt} &= \alpha_h(V) \cdot (1 - h) - \beta_h(V) \cdot h \end{aligned} \tag{B.2}$$

$$\alpha_n(V) = 0.01\frac{10-V}{e^{\frac{10-V}{10}}-1} \qquad \beta_n(V) = 0.125e^{\frac{-V}{80}}$$

$$\alpha_m(V) = 0.1\frac{25-V}{e^{\frac{25-V}{10}}-1} \qquad \beta_m(V) = 4e^{\frac{-V}{18}} \tag{B.3}$$

$$\alpha_h(V) = 0.07e^{\frac{-V}{20}} \qquad \beta_h(V) = \frac{1}{e^{\frac{30-V}{10}}+1}$$

To solve this model numerically, a solution for $\dot{n}$, $\dot{m}$, $\dot{h}$ and $\dot{V}$ must be found. Euler's method may then be applied to generate a discrete numerical model suitable for simulation on digital systems.

B.1 Gating Variable Equations - Steady State

It is shown in Appendix A that the activation and inactivation gating variables, stated in Equation set B.2, may all be written in the following form:

$$\frac{dx(t)}{dt} = \alpha_x \cdot (1 - x(t)) - \beta_x \cdot x(t) \tag{B.4}$$

In this form, V is assumed constant or suitably slow, such that α_x and β_x may be treated as constants. This form represents a 1D autonomous ordinary differential equation. In such equations, it may be shown that as $t \to \infty$, $x \to \pm\infty$ OR $x \to x_{ss}$. Since the gating variables are probabilities locked between 0 and 1 it may be safely assumed that as $t \to \infty$, x will approach a steady state value, x_{ss}.

At steady state, $dx/dt = 0$ by definition. Therefore:

$$\begin{aligned} 0 &= \alpha_x(1 - x_{ss}) - \beta_x x_{ss} \\ &= \alpha_x - \alpha_x x_{ss} - \beta_x x_{ss} \\ &= \alpha_x - x_{ss}(\alpha_x + \beta_x) \\ \therefore x_{ss} &= \frac{\alpha_x}{\alpha_x + \beta_x} \end{aligned} \tag{B.5}$$

This result is somewhat intuative since the equalibrium will only be reached when the proportion of active channels, x, complements the proportions of the switching rates, α_x and β_x.

B.2 Gating Variable Equations - Time Constant

Rearranging Equation B.4, it may be written in the form:

$$\begin{aligned}\frac{dx}{dt} &= \alpha_x(1-x) - \beta_x x \\ &= \alpha_x - \alpha_x x - \beta_x x \\ &= \alpha_x - x(\alpha_x + \beta_x)\end{aligned}$$

$$\frac{1}{\alpha_x + \beta_x}\frac{dx}{dt} = \frac{\alpha_x}{\alpha_x + \beta_x} - x$$

Using the newly defined steady state value in Equation B.5, this may be re-written as follows:

$$\frac{1}{\alpha_x + \beta_x}\frac{dx}{dt} = x_{ss} - x$$

Defining a scaling constant, τ_x, such that $\tau_x = 1/(\alpha_x + \beta_x)$, this may be re-phrased as:

$$\tau_x \frac{dx}{dt} = x_{ss} - x \tag{B.6}$$

In this form, τ_x represents the system time constant. This may be seen in the fact that it scales the rate of asymptotic approach without influencing the steady state value itself. For this reason the gating variable time constants may be defined as follows:

$$\tau_x = \frac{1}{\alpha_x + \beta_x} \tag{B.7}$$

B.3 Gating Variable Equations - Separation of Variables

As mentioned in Section B.1, it has been assumed that α_x and β_x are constant. If this is the case, x_{ss} and τ_x are also constant. This makes it possible to redefine Equation B.6 with respect to $x(t)$. First, rearranging the equation yields:

$$\frac{dx(t)}{x_{ss} - x(t)} = \frac{dt}{\tau_x}$$

Taking the integral of both sides:

$$\int \frac{1}{x_{ss} - x(t)} \cdot dx = \int \frac{1}{\tau_x} \cdot dt$$

$$-ln\,|x_{ss} - x(t)| = \frac{t}{\tau_x} + c$$

$$\implies x_{ss} - x(t) = e^{-\frac{t}{\tau_x}} e^c$$

$$\therefore x(t) = x_{ss} - ke^{-\frac{t}{\tau_x}} \tag{B.8}$$

where $k = e^c$. Setting $t = 0$, $x(t)$ becomes the initial condition, x_0, yielding:

$$x(0) = x_{ss} - ke^{-\frac{0}{\tau_x}}$$

$$x_0 = x_{ss} - k$$

$$\therefore k = x_{ss} - x_0$$

Using this definition in Equation B.8 results in the following definition for $x(t)$:

$$x(t) = x_{ss} - (x_{ss} - x_0)e^{-\frac{t}{\tau_x}} \tag{B.9}$$

In practice, x_{ss} and τ_x are not constant, instead dependant upon $V(t)$. If it is assumed that $V(t)$ (and therefore α_x and β_x) change slowly compared with the simulation sample rate, Equation B.9 may be re-written using Euler's method resulting in the following discrete representation:

$$x_{n+1} = x_{ss}(t_n) - (x_{ss}(t_n) - x_n) \cdot e^{-\frac{\Delta t}{\tau_x}} \tag{B.10}$$

B.4 Gating Variable Equations - Final Numerical Model

As identified at the beginning of Section B.1, each of the gating variables are written in the same form. As a result this process may be applied to each of the equations in

turn, resulting in the following numerical solutions:

$$\begin{aligned} n_{N+1} &= n_{ss}(t_N) - (n_{ss}(t_N) - n_N) \cdot e^{-\frac{\Delta t}{\tau_n}} \\ m_{N+1} &= m_{ss}(t_N) - (m_{ss}(t_N) - m_N) \cdot e^{-\frac{\Delta t}{\tau_m}} \\ h_{N+1} &= h_{ss}(t_N) - (h_{ss}(t_N) - h_N) \cdot e^{-\frac{\Delta t}{\tau_h}} \end{aligned} \tag{B.11}$$

where the steady state and time constant values are calculated using the associated α and β values from Equation set B.3 in the following equations:

$$\begin{aligned} x_{ss} &= \frac{\alpha_x}{\alpha_x + \beta_x} \\ \tau_x &= \frac{1}{\alpha_x + \beta_x} \end{aligned} \tag{B.12}$$

B.5 Membrane Voltage - Numerical Solution

The original Hodgkin-Huxley equation may be re-written in the channel generalised form:

$$\begin{aligned} C\frac{dV}{dt} &= \sum g_i(E_i - V) + I_{inj} \\ &= \sum g_i E_i - \sum g_i V + I_{inj} \end{aligned}$$

where g_i and E_i incrementally represent each of the considered channel conductances and Nernst potentials. Dividing through by $\sum g_i$ yields:

$$\underbrace{\frac{C}{\sum g_i}}_{\tau_V}\frac{dV}{dt} = \underbrace{\frac{g_i E_i}{\sum g_i} + \frac{I_{inj}}{\sum g_i}}_{V_{ss}} - V$$

This equation is in the same form as that of Equation B.6, resulting in the following definitions for τ_V and V_{ss}:

$$\begin{aligned} V_{ss} &= \frac{g_i E_i}{\sum g_i} + \frac{I_{inj}}{\sum g_i} \\ \tau_V &= \frac{C}{\sum g_i} \end{aligned} \tag{B.13}$$

Again, assuming that V (and therefore $\sum g_i$) changes slowly with respect to the sample rate, Euler's method may be applied to generate a discreet numerical solution, resulting in the following equation:

$$V_{n+1} = V_{ss}(t_n) - (V_{ss}(t_n) - V_n) \cdot e^{-\frac{\Delta t}{\tau_V}} \tag{B.14}$$

Appendix C

Linear Piecewise Minimisation

Each line in the point-to-point linear piecewise approximation of $y = 2^x$ passes through the points $(x_1, 2^{x_1})$ and $(x_2, 2^{x_2})$, where x_1 and x_2 are the boundary locations for the approximation region. A single line may therefore be defined in the following form:

$$l_n = a_n x + b_n \tag{C.1}$$

where a_n is the gradient and b_n is the offset of the n-th line, l_n. The gradient and offset values are defined by the linear interpolation of the two boundary points (x_1, y_1) and (x_2, y_2) across the valid region and may be calculated as follows:

$$\begin{aligned} a &= \frac{y_2 - y_1}{x_2 - x_1} \\ b &= y_2 - a x_2 \end{aligned} \tag{C.2}$$

The absolute error for an approximation of $y = 2^x$ may be defined within each valid region as:

$$E_{Abs} = |ax + b - 2^x| \tag{C.3}$$

In such an arrangement, the maximum error of the approximation, E_{max}, will occur between x_1 and x_2 at the location $x = x_{max}$, while the error at each boundary will be zero. Adding an offset to the line allows this error to be shared between the boundaries and the maximum error location. As the offset is increased, the error at the boundaries increases linearly, while the error in the maximum error location decreases linearly. The minimum maximum error will therefore occur when:

$$-E_{x_1} = E_{x_{max}}$$

Therefore:

$$-(ax_1 + b + c - 2^{x_1}) = ax_{max} + b + c - 2^{x_{max}}$$

$$2c = -(ax_{max} + b - 2^{x_{max}}) - (ax_1 + b - 2^{x_1})$$

By definition, $ax_1 + b = 2^{x_1}$ and $ax_{max} + b - 2^{x_{max}} = E_{x_{max}}$, meaning:

$$2c = E_{x_{max}}$$

$$c = \frac{1}{2}E_{x_{max}}$$

The line that best spreads the error between the boundaries and maximum error location is therefore defined as follows:

$$l_n = a_n x + b_n - \frac{1}{2}E_n \tag{C.4}$$

where E_n is the linear approximations maximum error without any offset correction.

Appendix D

Python Neural Network Class

The following neural-network class was written using Python to support arbitrary activation functions. It uses the 'activation' function library defined in Appendix E.

D.1 Python Class Library - 'neuralFunctions.py'

```
# Import numpy for easy array handling and dot products
import numpy
# Import scipy.special for the sigmoid function expit()
import scipy.special
# Import plotting library
import matplotlib.pyplot as plt
# Import custom activation functions library
import activation
# Import pickle for saving objects (the neural network)
import dill as pickle
import copy

# Use something like:...
# n = LoadNeuralNetwork("./RESULTS/2017-12-18_10-09-13_MNIST/Net.p")
def LoadNeuralNetwork(filename):
    # Loads a neural network from file
```

```
17      with open(filename, 'rb') as f:
18          n = pickle.load(f)
19      return n
20
21  # Neural network class definition
22  class NeuralNetwork:
23      # Initialize the network
24      def __init__(self, shape, learning_rate, activation_function,
25        bias = 1.0, filename="", std="", weights=""
26      ):
27          # Set the number of nodes in each input, hidden and output layer
28          self.shape = shape
29          self.layers = len(shape)
30          self.bias = bias
31
32          # Weight matrices, wih (input->hidden) and who (hidden->output)
33          # Weight inside the matrices are w_i_j
34          # Where a link is from node i to node j in the next layer
35          if ( weights =="" ):
36              if ( std=="" ):
37                  self.weights = [
38                    numpy.random.normal(0.0, pow(self.shape[i+1], -0.5),
39                    (self.shape[i+1], self.shape[i]+1))
40                    for i in range(self.layers-1)
41                  ]
42              else:
43                  self.weights = [numpy.random.normal(0.0, std,
44                    (self.shape[i+1], self.shape[i]+1))
45                    for i in range(self.layers-1)
46                  ]
47          else:
48              self.weights = copy.deepcopy(weights)
49
50          # Set the learning rate
51          self.lr = learning_rate
52          # Set the activation function
53          self.activation_function = lambda x: activation_function(x)
```

```
54
55      def save(self, filename):
56          # Saves the neural network to file
57          with open(filename, 'wb') as f:
58              pickle.dump(self, f, pickle.HIGHEST_PROTOCOL)
59
60      # Train the network using back-propagation of errors
61      def train(self, inputs_list, targets_list):
62          # Add bias onto inputs_list
63          inputs_list = numpy.append(inputs_list,self.bias)
64          targets_list = numpy.append(targets_list,self.bias)
65          # Convert inputs into 2D arrays
66          layer_outputs = [numpy.array(inputs_list, ndmin=2).T]
67          targets_array = numpy.array(targets_list, ndmin=2).T
68
69          layer_inputs = []
70
71          for i in range(self.layers - 1):
72              # Calculate signals into layer (LHS of weights)
73              layer_inputs.append([numpy.dot(self.weights[i],
74                layer_outputs[i])])
75
76              # Calculate the signals from layer (RHS of weights)
77              ac_results = numpy.append(self.activation_function(
78                layer_inputs[i][0]), self.bias)
79              ac_results = numpy.array(ac_results, ndmin=2).T
80
81              layer_outputs.append(ac_results)
82
83          # Current error is (target - actual)
84          errors = [targets_array - layer_outputs[self.layers-1]]
85
86          # Hidden layer errors are the output errors, ...
87          # ...split by the weights, recombined at hidden nodes
88          for i in range(2, self.layers):
89              layer_error = numpy.dot(numpy.asfarray(
90                self.weights[self.layers - i].T), errors[i-2][0:-1])
```

```
91              errors.append(layer_error)
92
93          errors.reverse()
94
95          # Update the weights for the links between the layers
96          for i in range(self.layers - 1):
97              # Calculate the new weights for layer
98              gradient = layer_outputs[i+1] * (1.0 - layer_outputs[i+1])
99              new_weights = self.lr * numpy.dot((errors[i] * gradient),
100               numpy.transpose(layer_outputs[i]))
101
102             # Remove the weights going into the bias nodes...
103             # ...as bias node is fixed at self.bias
104             self.weights[i] += new_weights[0:-1]
105
106     # Query the network
107     def query(self, inputs_list):
108         # Add bias onto inputs_list
109         inputs_list = numpy.append(inputs_list,self.bias)
110         # Convert the inputs list into a 2D array
111         # First entry is the output of the input layer!
112         layer_outputs = [numpy.array(inputs_list, ndmin=2).T]
113
114         # Layer inputs are the output side of the weights layers
115         layer_inputs = []
116
117         for i in range(self.layers - 1):
118             # Calculate signals into layer (LHS of weights)
119             layer_inputs.append([numpy.dot(self.weights[i],
120               layer_outputs[i])])
121
122             # Calculate the signals from layer (RHS of weights)
123             ac_results = numpy.append(self.activation_function(
124               layer_inputs[i][0]), self.bias)
125             ac_results = numpy.array(ac_results, ndmin=2).T
126
127             layer_outputs.append(ac_results)
```

```
128
129         return layer_outputs[self.layers - 1][0:-1]
```

Appendix E

Activation Function Library for Python

Each of the new activation functions introduced in this thesis were first written and tested using a custom made C library for Python. This allowed for implementations that were closer to their hardware equivalents, with the editing and moving of individual bits. The files used to build this library are shown below for reference.

E.1 Header File - 'activationmodule.py'

This file registers the library with Python. The library may be built and installed by calling `python3 activationmodule.py build` in terminal, followed by the command `sudo python3 activationmodule.py install`. Scripts may then access the library, using `import activation` whenever they require access to the activation functions.

```
from distutils.core import setup, Extension

module1 = Extension('activation',
        sources = ['activationmodule.c'])

setup (name = 'Activation',
        version = '1.0',
        description = 'A set of activation functions',
        ext_modules = [module1])
```

E.2 Library Functions - 'activationmodule.c'

This file defines the C functions used by the activation function library. Also included are the python specific requisites to enable the interface between a python script and these C functions.

```c
#include <Python.h>
#include <math.h>

#define FRACT_BITS 23
#define FIXED_ONE (1 << FRACT_BITS)
#define INT2FIXED(x) ((x) << FRACT_BITS)
#define FLOAT2FIXED(x) ((int)((x) * FIXED_ONE))
#define FIXED2INT(x) ((x) >> FRACT_BITS)
#define FIXED2DOUBLE(x) (((double)(x)) / FIXED_ONE)
#define FIXED2FLOAT(x) (((float)(x)) / FIXED_ONE)

#define MANTISSA_ONE (1 << 23)
#define MANTISSA2FLOAT(x) (((float)(x)) / MANTISSA_ONE)
#define FLOAT2MANTISSA(x) ((unsigned int)((x) * MANTISSA_ONE))

// ------------------- Define Datatypes -------------------------

typedef union fixed {
    struct {
        unsigned int decimal : 23;
        int integer          : 6;
        int waste            : 3;
    } part;
    int full;
} FIXED;

union FloatingPointIEEE754 {
    struct {
        unsigned int mantissa : 23;
        unsigned int exponent : 8;
        unsigned int sign     : 1;
```

```
32      } raw;
33      float f;
34  };
35
36  // ------------------- Main Activation functions --------------------
37
38  // TwoXSig Function - Base two sigmoid function
39  static PyObject * activation_twoxsig(PyObject *self, PyObject *args)
40  {
41      union FloatingPointIEEE754 i;
42      float power;
43      float res;
44
45      if (!PyArg_ParseTuple(args, "f", &i.f)) {
46          return NULL;
47      }
48
49      power = powf(2, -i.f);
50      res = 1/(1+power);
51      return Py_BuildValue("f", res);
52  }
53
54  // One-line base two sigmoid approximation
55  static PyObject * activation_twoxsig_1_line(PyObject *self,
56  PyObject *args)
57  {
58      union FloatingPointIEEE754 i;
59      union FloatingPointIEEE754 power;
60      float res;
61      FIXED i_fixed;
62
63      if (!PyArg_ParseTuple(args, "f", &i.f)) {
64          return NULL;
65      }
66
67      // Invert X since 2^-x is needed!
68      i.f = -i.f;
```

```
69
70      // If number is huge (i.e. exponent bigger than 31)
71      if (i.raw.exponent > 131) {
72          // ... asymptote will have been reached.
73          res = i.raw.sign;
74      } else {
75          // Convert to fixed representation
76          i_fixed.full = FLOAT2FIXED(i.f);
77          // Result of 2^x is always positive!
78          power.raw.sign = 0;
79          // Add bias to integer value
80          power.raw.exponent = 127+i_fixed.part.integer;
81
82          // If the mantissa is at risk of underflowing...
83          if (MANTISSA2FLOAT(i_fixed.part.decimal) < (0.02982/0.97018)) {
84              // ...reduce the exponent by one...
85              power.raw.exponent = power.raw.exponent - 1;
86              // ... and multiply the decimal part by 2.
87              power.raw.mantissa = FLOAT2MANTISSA(
88                (0.97018*MANTISSA2FLOAT(i_fixed.part.decimal)+0.97018)*2.0
89              );
90          } else {
91              // ...move decimal value into mantissa with correction.
92              power.raw.mantissa = FLOAT2MANTISSA(
93                0.97018*MANTISSA2FLOAT(i_fixed.part.decimal) - 0.02982
94              );
95          }
96
97          res = 1/(1+power.f);
98      }
99      return Py_BuildValue("f", res);
100 }
101
102 // Bitwise approximation of base two model
103 static PyObject * activation_twoxsig_basic(PyObject *self,
104 PyObject *args)
105 {
```

```
106     union FloatingPointIEEE754 i;
107     union FloatingPointIEEE754 power;
108     float res;
109     FIXED i_fixed;
110 
111     if (!PyArg_ParseTuple(args, "f", &i.f)) {
112         return NULL;
113     }
114 
115     // Invert X since 2^-x is needed
116     i.f = -i.f;
117     // If number is huge (i.e. exponent bigger than 31)...
118     if (i.raw.exponent > 131) {
119         // ... asymptote will have been reached.
120         res = i.raw.sign;
121     } else {
122         i_fixed.full = FLOAT2FIXED(i.f);
123         // Result of 2^x is always positive
124         power.raw.sign = 0;
125         // Add bias to integer value
126         power.raw.exponent = 127+i_fixed.part.integer;
127         // Move decimal value into mantissa
128         power.raw.mantissa = i_fixed.part.decimal;
129         res = 1/(1+power.f);
130     }
131 
132     return Py_BuildValue("f", res);
133 }
134 
135 // ------------------- Python Specifics ----------------------
136 
137 // List of functions defined in the module
138 static PyMethodDef activation_methods[] = {
139     {"twoxsig",
140      activation_twoxsig,
141      METH_VARARGS,
142      PyDoc_STR("A full 2^x sigmoid function.")
```

```
143     },
144     {"twoxsig_1_line",
145      activation_twoxsig_1_line,
146      METH_VARARGS,
147      PyDoc_STR("A 1-line approximation of the 2^x sigmoid function.")
148     },
149     {"twoxsig_basic",
150      activation_twoxsig_basic,
151      METH_VARARGS,
152      PyDoc_STR("An efficient bit-bashing approximation of \
153        the 2^x sigmoid function.")
154     },
155     {NULL, NULL}  // sentinel
156 };
157 
158 // Define module documentation
159 PyDoc_STRVAR(module_doc,
160     "A set of efficent and fast activation functions, \
161       by Jonathan G-H-Cater."
162 );
163 
164 // Define module entry
165 static struct PyModuleDef Activations =
166 {
167     PyModuleDef_HEAD_INIT,
168     "activation", // Link Module Name
169     module_doc, // Link Module Documentation
170     -1,
171     activation_methods
172 };
173 
174 // Initialization function for the module
175 PyMODINIT_FUNC PyInit_activation(void)
176 {
177     return PyModule_Create(&Activations);
178 }
```